WebSphere® Application Server Network Deployment for IBM i, Version 7.0

Scripting the application serving environment

Note

Before using this information, be sure to read the general information under “Notices” on page 1327.

Compilation date: October 9, 2008

Contents

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback** .
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-5250.

  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Changes to serve you more quickly

Print sections directly from the information center navigation

PDF books are provided as a convenience format for easy printing, reading, and offline use. The information center is the official delivery format for IBM WebSphere Application Server documentation. If you use the PDF books primarily for convenient printing, it is now easier to print various parts of the information center as needed, quickly and directly from the information center navigation tree.

To print a section of the information center navigation:

1. Hover your cursor over an entry in the information center navigation until the **Open Quick Menu** icon is displayed beside the entry.
2. Right-click the icon to display a menu for printing or searching your selected section of the navigation tree.
3. If you select **Print this topic and subtopics** from the menu, the selected section is launched in a separate browser window as one HTML file. The HTML file includes each of the topics in the section, with a table of contents at the top.
4. Print the HTML file.

For performance reasons, the number of topics you can print at one time is limited. You are notified if your selection contains too many topics. If the current limit is too restrictive, use the feedback link to suggest a preferable limit. The feedback link is available at the end of most information center pages.

Under construction!

The Information Development Team for IBM WebSphere Application Server is changing its PDF book delivery strategy to respond better to user needs. The intention is to deliver the content to you in PDF format more frequently. During a temporary transition phase, you might experience broken links. During the transition phase, expect the following link behavior:

- Links to Web addresses beginning with http:// work
- Links that refer to specific page numbers within the same PDF book work
- The remaining links will *not* work. You receive an error message when you click them

Thanks for your patience, in the short term, to facilitate the transition to more frequent PDF book updates.

Chapter 1. Using scripting (wsadmin)

The WebSphere® administrative (wsadmin) scripting program is a powerful, non-graphical command interpreter environment enabling you to run administrative operations in a scripting language.

About this task

The wsadmin tool is intended for production environments and unattended operations. You can use the wsadmin tool to perform the same tasks that you can perform using the administrative console.

The following list highlights the topics and tasks available with scripting:

- Getting started with scripting Provides an introduction to WebSphere Application Server scripting and information about using the wsadmin tool. Topics include information about the scripting languages and the scripting objects, and instructions for starting the wsadmin tool.
- Using the script library to automate the application serving environment Provides a set of Jython script procedures that automate the most common application server administration functions. For example, you can use the script library to easily configure servers, applications, mail settings, resources, nodes, business-level applications, clusters, authorization groups, and more. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.
- Deploying applications Provides instructions for deploying and uninstalling applications. For example, stand-alone Java™ archive files and Web archive files, the administrative console, remote Enterprise Archive (EAR) files, file transfer applications, and so on.
- Managing deployed applications Includes tasks that you perform after the application is deployed. For example, starting and stopping applications, checking status, modifying listener address ports, querying application state, configuring a shared library, and so on.
- Configuring servers Provides instructions for configuring servers, such as creating a server, modifying and restarting the server, configuring the Java virtual machine, disabling a component, disabling a service, and so on.
- Configuring connections to Web servers Includes topics such as regenerating the plug-in, creating new virtual host templates, modifying virtual hosts, and so on.
- Managing servers Includes tasks that you use to manage servers. For example, stopping nodes, starting and stopping servers, querying a server state, starting a listener port, and so on.
- Clustering servers Includes topics about clusters, such as creating clusters, creating cluster members, querying a cluster state, removing clusters, and so on.
- Configuring security Includes security tasks, for example, enabling and disabling administrative security, enabling and disabling Java 2 security, and so on.
- Configuring data access Includes topics such as configuring a Java DataBase Connectivity (JDBC) provider, defining a data source, configuring connection pools, and so on.
- Configuring messaging Includes topics about messaging, such as Java Message Service (JMS) connection, JMS provider, WebSphere queue connection factory, MQ topics, and so on.
- Configuring mail, URLs, and resource environment entries Includes topics such as mail providers, mail sessions, protocols, resource environment providers, referenceables, URL providers, URLs, and so on.
- Troubleshooting Provides information about how to troubleshoot using scripting. For example, tracing, thread dumps, profiles, and so on.
- Scripting reference material Includes all of the reference material related to scripting. Topics include the syntax for the wsadmin tool and for the administrative command framework, explanations and examples for all of the scripting object commands, the scripting properties, and so on.

Chapter 2. Getting started with scripting

Scripting is a non-graphical alternative that you can use to configure and manage WebSphere Application Server.

About this task

The WebSphere Application Server wsadmin tool provides the ability to run scripts. The wsadmin tool supports a full range of product administrative activities.

The following figure illustrates the major components involved in a wsadmin scripting solution:

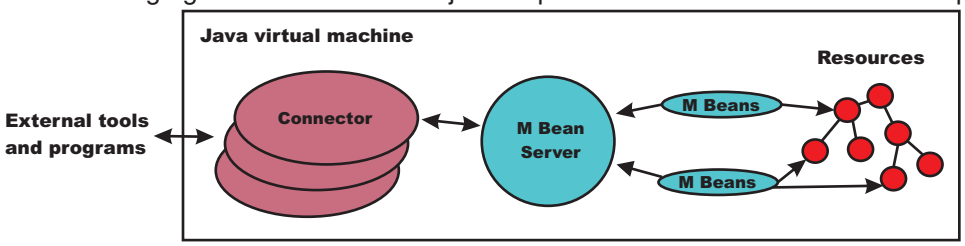

Figure 1: A WebSphere Application Server scripting solution

The wsadmin tool supports two scripting languages: Jacl and Jython. Five objects are available when you use scripts:

- **AdminControl**: Use to run operational commands.
- **AdminConfig**: Use to run configurational commands to create or modify WebSphere Application Server configurational elements.
- **AdminApp**: Use to administer applications.
- **AdminTask**: Use to run administrative commands.
- **Help**: Use to obtain general help.

The scripts use these objects to communicate with MBeans that run in WebSphere Application Server processes. MBeans are Java objects that represent Java Management Extensions (JMX) resources. JMX is an optional package addition to Java 2 Platform Standard Edition (J2SE). JMX is a technology that provides a simple and standard way to manage Java objects.

To perform a task using scripting, you must first perform the following steps:

1. Choose a scripting language. The wsadmin tool only supports Jacl and Jython scripting languages. Jacl is the language specified by default. If you want to use the Jython scripting language, use the -lang option or specify it in the wsadmin.properties file.
2. Start the wsadmin scripting client interactively, as an individual command, in a script, or in a profile.

What to do next

Before you perform any task using scripting, make sure that you are familiar with the following concepts:

- Java Management Extensions (JMX)
- WebSphere Application Server configuration model
- wsadmin tool
- Jacl syntax or Jython syntax
- Scripting objects

Optionally, you can customize your scripting environment. For more information, see Scripting environment properties.

After you become familiar with the scripting concepts, choose a scripting language, and start the scripting client, you are ready to perform tasks using scripting.

What is new for scripted administration (wsadmin)

This topic highlights what is new or changed, for users who are going to customize, administer, monitor, and tune production server environments using the wsadmin tool.

Deprecated, stabilized, and removed features describes features that are being replaced or removed in this or future releases.

Improved administrative scripting features

Jython script library

The Jython script library provides a set of procedures to automate the most common application server administration functions. For example, you can use the script library to easily configure servers, applications, mail settings, resources, nodes, business-level applications, clusters, authorization groups, and more. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

For more information, see Chapter 3, “Using the script library to automate the application serving environment,” on page 81.

Wildcard pattern usage for the AdminTask help and AdminConfig list commands

Use Java regular expression patterns and wildcard patterns with the AdminTask.help('–commands') command to query for command names and with AdminConfig.list, AdminConfig.types and AdminConfig.listTemplates functions to query for configuration object types.

For more information about using patterns with the AdminTask object, see “Obtaining online help using scripting” on page 59

For more information about using patterns with the AdminConfig object, see “Commands for the AdminConfig object” on page 1199

Complex parameter type support

Use primitive and complex Java™ data types for the parameters of the AdminTask commands.

For more information, see “Data types for the AdminTask object” on page 76

AdminTask command support for configuring certificate authority (CA) clients

Use the wsadmin tool to manage certificate authority (CA) client configurations. Use the commands and parameters in the CAClientCommands group to create, modify, query, and remove connections to a third-party CA server.

For more information, see “CAClientCommands command group for the AdminTask object” on page 645

AdminTask command support for configuring security auditing

While security authentication and authorization ensures that users must have access to view protected resources, security auditing provides a mechanism to validate the integrity of a security computing environment. Security auditing collects and logs authentication, authorization, system management, security, and audit policy events in audit event records. You can analyze audit event records to determine possible security breaches, threats, attacks, and potential weaknesses in the security configuration of your environment. Enable security auditing in your environment.

For more information, see “Configuring security auditing using scripting” on page 732

AdminTask command support for configuring multiple security domains

Create multiple security configurations and assign them to different applications in WebSphere Application Server processes. By creating multiple security domains, you can configure different security attributes for both administrative and user applications within a cell environment. You can configure different applications to use different security configurations by assigning the servers or clusters or SIBuses that host these applications to the security domains. Only users assigned to the administrator role can configure multiple security domains.

For more information, see “Configuring multiple security domains using scripting” on page 552

AdminTask command support for configuring business-level applications

A business-level application is an administration model that provides the entire definition of an application as it makes sense to the business. A business-level application is a WebSphere® configuration artifact, similar to a server or cluster, that is stored in the product configuration repository.

For more information, see “Setting up business-level applications using scripting” on page 211

AdminTask command support for configuring a flexible management environment

Create a flexible management environment to locally or remotely submit and manage administrative jobs. You can use the job manager to manage applications, modify configurations, and control the application server runtime.

For more information, see “Administering jobs in a flexible management environment using scripting” on page 401

AdminTask command support for managing configurations using properties files

Manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

For more information, see “PropertiesBasedConfiguration command group for the AdminTask object” on page 475

AdminTask command support for configuring the WebSphere® DataPower® appliance manager

WebSphere® DataPower® appliance manager provides a set of capabilities for managing sets of appliances. DataPower appliance manager can be used to manage appliances with a 3.6.0.4 or higher level of firmware. IBM® WebSphere® DataPower SOA Appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments.

For more information, see "Setting up the DataPower appliance manager using scripting" on page 358

AdminTask command support for configuring proxy servers

Use the commands and parameters in the ProxyManagement group to configure proxy servers for Web modules.

For more information, see "ProxyManagement command group for the AdminTask object" on page 325

Overview and new features for scripting the application serving environment

Use the links provided in this topic to learn about the administrative features.

"What is new for scripted administration (wsadmin)" on page 4

This topic provides an overview of new and changed features for administrative scripting and the wsadmin tool.

Introduction: Administrative scripting (wsadmin)

This topic provides an introduction to administrative scripting and the wsadmin tool.

Java Management Extensions (JMX)

Java Management Extensions (JMX) is a framework that provides a standard way of exposing Java resources, for example, application servers, to a system management infrastructure. Using the JMX framework, a provider can implement functions, such as listing the configuration settings, and editing the settings. This framework also includes a notification layer that management applications can use to monitor events such as the startup of an application server.

JMX key features

The key features of the WebSphere Application Server Version 6 implementation of JMX include:

- All processes that run the JMX agent.
- All run-time administration that is performed through JMX operations.
- Connectors that are used to connect a JMX agent to a remote JMX-enabled management application. The following connectors are supported:
  - SOAP JMX Connector
  - JMX Remote application programming interface (JSR 160) Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector, (the JSR160RMI connector)
  - Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector
  - Inter-Process Communications (IPC)
- Protocol adapters that provide a management view of the JMX agent through a given protocol. Management applications that connect to a protocol adapter are usually specific to a given protocol.
- The ability to query and update the configuration settings of a run-time object.

- The ability to load, initialize, change, and monitor application components and resources during run-time.

JMX architecture

The JMX architecture is structured into three layers:

- Instrumentation layer - Dictates how resources can be wrapped within special Java beans, called managed beans (MBeans).
- Agent layer - Consists of the MBean server and agents, which provide a management infrastructure. The services that are implemented include:
  - Monitoring
  - Event notification
  - Timers
- Management layer - Defines how external management applications can interact with the underlying layers in terms of protocols, APIs, and so on. This layer uses an implementation of the distributed services specification (JSR-077), which is not yet part of the Java 2 platform, Enterprise Edition (J2EE) specification.

The layered architecture of JMX is summarized in the following figure:

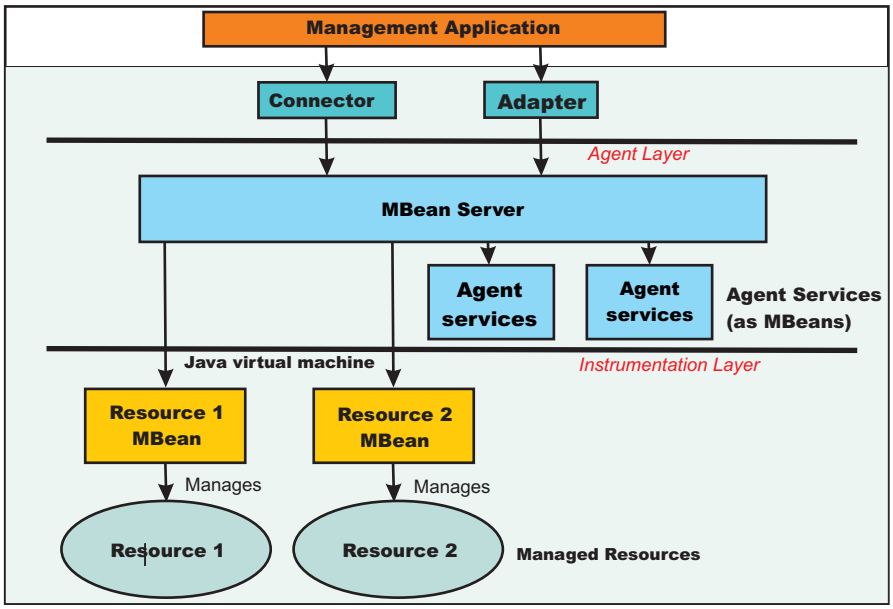

Figure 1: JMX architecture

JMX distributed administration

The following figure shows how the JMX architecture fits into the overall distributed administration topology of a Network Deployment environment:

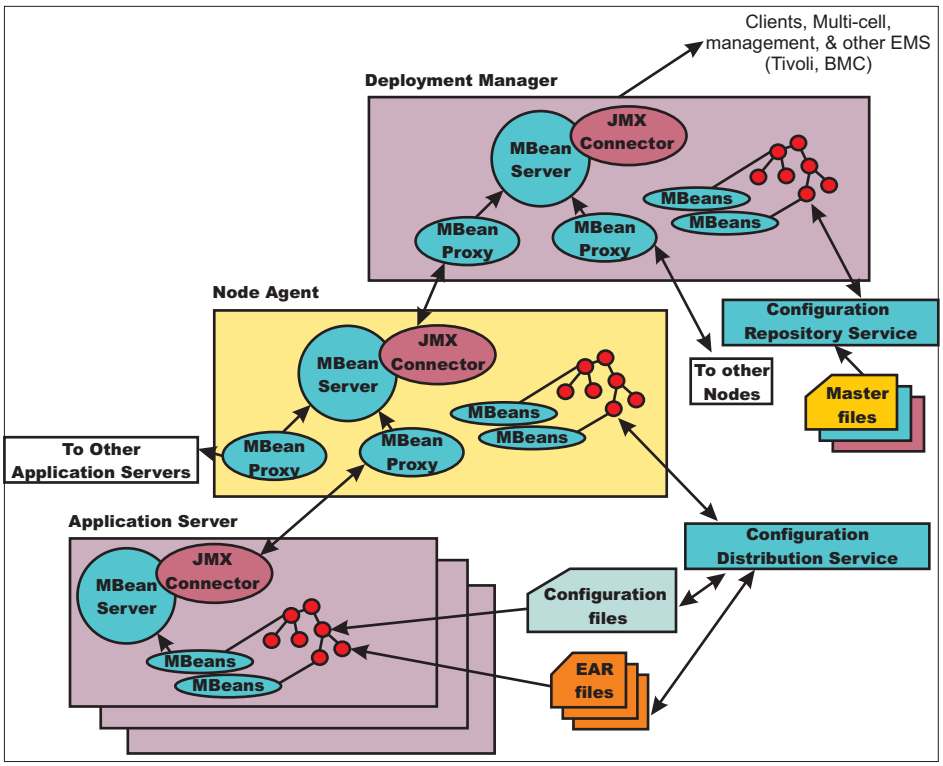

Figure 2: WebSphere Application Server distributed administration of JMX

The key points of this distributed administration architecture include:

- Internal MBeans that are local to the Java virtual machine (JVM) register with the local MBean server.
- External MBeans have a local proxy to their MBean server. The proxy registers with the local MBean server.

  Using the MBean proxy the local MBean server can pass the message to an external MBean server that is located on:

  – A node agent that has an MBean proxy for all the servers within its node. The MBean proxies for other nodes are not used.

  – The deployment manager has MBean proxies for all the node agents in the cell.

JMX Mbeans

WebSphere Application Server provides a number of MBeans, each of which has different functions and operations available. For example, an application server MBean can expose operations such as start and stop. An application MBean can expose operations such as install and uninstall. Some JMX usage scenarios that you can encounter include:

- External programs that are written to control the Network Deployment run time and its WebSphere resources by programmatically accessing the JMX API.
- Third-party applications that include custom JMX MBeans as part of the deployed code, supporting the JMX API management of application components and resources.

The following example illustrates how to obtain the name of a particular MBean:

Using Jacl:

```
set am [$AdminControl queryNames type=ApplicationManager,process=server1,*]
```

Using Jython:

```
am = AdminControl.queryNames('type=ApplicationManager,process=server1,*')
```

Each WebSphere Application Server runtime MBean can have attributes, operations, and notifications. The complete documentation for each MBean that is supplied with WebSphere Application Server is available in an HTML table that is installed in each copy of the WebSphere Application Server product. Under the main installation directory for the product, there is the `web` directory. Under the `web` directory there is another directory called `mbeanDocs`. In the `mbeanDocs` directory there are several HTML files; one HTML file for each MBean supplied with WebSphere Application Server. There is also an `index.html` file that ties all the individual MBean files together in a top-level navigation tree. Each MBean provides a summary of its attributes, operations, and notifications.

JMX benefits

The use of JMX for management functions in WebSphere Application Server provides the following benefits:

- Enables the management of Java applications without significant investment.
- Relies on a core-managed object server that acts as a management agent.
- Java applications can embed a managed object server and make some of its functionality available as one or several MBeans that are registered with the object server.
- Provides a scalable management architecture.
- Every JMX agent service is an independent module that can be plugged into the management agent.
- The API is extensible, allowing new WebSphere Application Server and custom application features to be easily added and exposed through this management interface.
- Integrates existing management solutions.
- Each process is self-sufficient when it comes to the management of its resources. No central point of control exists. In principle, a JMX-enabled management client can be connected to any managed process and interact with the MBeans that are hosted by that process.
- JMX provides a single, flat, domain-wide approach to system management. Separate processes interact through MBean proxies that support a single management client to seamlessly navigate through a network of managed processes.
- Defines the interfaces that are necessary for management only.
- Provides a standard API for exposing application and administrative resources to management tools.

WebSphere Application Server configuration model

Understanding the relationship between the different configuration objects is essential when creating wsadmin scripts that perform configuration function.

Configuration data is stored in several different XML files which the server run time reads when it starts and responds to the component settings stored there. The configuration data includes the settings for the run time, such as, Java virtual machine (JVM) options, thread pool sizes, container settings, and port numbers the server will use. Other configuration files define Java 2 Platform, Enterprise Edition (J2EE) resources to which the server connects in order to obtain data that is needed by the application logic. Security settings are stored in a separate document from the server and resource configuration. Application-specific configuration, such as, deployment target lists, session configuration, and cache settings, are stored in files under the root directory of each application. When viewing the XML data in the configuration files, you can discern relationship between the configuration objects.

For more information on the WebSphere Application Server configuration objects view the HTML tables in the *installroot*`/web/configDocs` directory. There are several subdirectories, one for each configuration package in the model. The `index.html` file ties all of the individual configuration packages together in a top-level navigation tree. Each configuration package lists the supported configuration classes and the configuration class lists all of the supported properties. The properties with names that end with the at (`@`)

character imply that property is a reference to a different configuration object within the configuration data. The properties with names that end with an asterisk (*) character imply that the property is a list of other configuration objects.

Jacl

Jacl is an alternate implementation of TCL, and is written entirely in Java code.

The wsadmin tool uses Jacl V1.3.2.

Deprecation of the Jacl syntax in the wsadmin tool

Deprecation of a product feature does not mean that the feature is removed from the product immediately. Deprecation is a process of announcing the intent to remove the feature at some future time. The WebSphere Application Server deprecation procedure calls for a feature to remain in the product for two full release cycles before the feature can be removed. For more information about deprecation of features, see the Deprecated and removed features article in the *Migrating, coexisting, and interoperating* PDF.

The wsadmin administrative scripting program supports two scripting languages, Jacl and Jython. The Version 6.1 release of WebSphere Application Server marked the start of the deprecation process for the Jacl syntax that is associated with the wsadmin tool. The Jacl syntax for the wsadmin tool continues to remain in the product and is supported for at least one major product release after Version 7.0. After that time, the Jacl language support might be removed from the wsadmin tool.

The Jython syntax for the wsadmin tool is the strategic direction for WebSphere Application Server administrative automation. WebSphere Application Server continues to provide enhanced administrative functions and tooling that support product automation and the use of the Jython syntax. The following Jython scripting-related enhancements are provided in WebSphere Application Server:

- Administrative console command assist - A new feature of the WebSphere Application Server administrative console that displays the wsadmin command that is equivalent to the action taken by the user that interacts with the console. The output from the console command assist feature can be transferred directly to the WebSphere Application Server Tool, which simplifies the development of Jython scripts that are based on administrative console actions. You can also save the output after using the console command assist feature in a plain text file for later use.
- Jacl-to-Jython conversion utility - a program that converts Jacl syntax wsadmin scripts into equivalent Jython syntax wsadmin scripts. Dozens of new wsadmin high-level commands that decouple the script from the underlying administrative model through use of simple parameters and smart default logic.

All future enhancements in the area of WebSphere Application Server scripting will focus on use of the Jython syntax. While Jacl will remain as a component that is shipped with WebSphere Application Server for at least one additional full release, no new tooling or explicit enhancements will be created for the Jacl syntax.

Basic syntax:

The basic syntax for a Jacl command is the following:

```
Command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a Jacl procedure. For example:

```
puts stdout {Hello, world!}
=> Hello, world!
```

In this example, the command is **puts** which takes two arguments, an I/O stream identifier and a string. The **puts** command writes the string to the I/O stream along with a trailing new line character. The arguments are interpreted by the command. In the example, `stdout` is used to identify the standard output

stream. The use of `stdout` as a name is a convention employed by the **puts** command and the other I/O commands. `stderr` identifies the standard error output, and `stdin` identifies the standard input.

Variables

The **set** command assigns a value to a variable. This command takes two arguments: the name of the variable and the value. Variable names can be any length and are case sensitive. You do not have to declare Jacl variables before you use them. The interpreter will create the variable when it is first assigned a value. For example:

```
set a 5
=> 5
set b $a
=> 5
```

The second example assigns the value of variable `a` to variable `b`. The use of dollar sign ($) is indicates variable substitution. You can delete a variable with the **unset** command, for example:

```
unset varName1 varName2 ...
```

You can pass any number of variables to the **unset** command. The **unset** command will give error if a variable is not already defined. You can delete an entire array or just a single array element with the **unset** command. Using the **unset** command on an array is a easy way to clear out a big data structure. The existence of a variable can be tested with the **info exists** command. You may have to test for the existence of the variable because the `incr` parameter requires that a variable exist first, for example:

```
if ![info exists foobar] {set foobar 0} else {incr foobar}
```

Command substitution:

The second form of substitution is command substitution. A nested command is delimited by square brackets, [ ]. The Jacl interpreter evaluates everything between the brackets and evaluates it as a command. For example:

```
set len [string length foobar]
=> 6
```

In this example, the nested command is the following: `string length foobar`. The **string** command performs various operations on strings. In this case, the command asks for the length of the string `foobar`. If there are several cases of command substitution within a single command, the interpreter processes them from left bracket to right bracket. For example:

```
set number "1 2 3 4"
=> 1 2 3 4
set one [lindex $number 0]
=> 1
set end [lindex $number end]
=> 4
set another {123  456  789}
=> 123  456  789
set stringLen [string length [lindex  $another  1]]
=> 3
set listLen [llength [lindex $another 1]
=> 1
```

Math expressions:

The Jacl interpreter does not evaluate math expressions. Use the **expr** command to evaluate math expressions. The implementation of the **expr** command takes all arguments, concatenates them into a single string, and parses the string as a math expression. After the **expr** command computes the answer, it his formatted into a string and returned. For example:

```
expr 7.2 / 3
=> 2.4
```

Backslash substitution:

The final type of substitution done by the Jacl interpreter is backslash substitution. Use this to quote characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. If you are using lots of backslashes, instead you can group things with curly braces to turn off all interpretation of special characters. There are cases where backslashes are required. For example:

```
set dollar "This is a string \$contain dollar char"
=> This is a string $contain dollar char

set x $dollar
=> This is a string $contain dollar char

set  group {$ {} [] { [ } ]}
=> $ {} [] { [ } ]
```

You can also use backslashes to continue long commands on multiple lines. A new line without the backslash terminates a command. A backslashes that are the last character on a line convert into a space. For example:

```
set  totalLength  [expr  [string  length  "first string"] + \
[string  length  "second string"]]
=> 25
```

Grouping with braces and double quotes:

Use double quotes and curly braces to group words together. Quotes allow substitutions to occur in the group and curly braces prevent substitution. This rule applies to command, variable, and backslash substitutions. For example:

```
set s Hello
=> Hello

puts stdout "The length of $s is [string length $s]."
=> The length of Hello is 5.

puts  stdout  {The length of $s is [string  length  $s].}
=> The length of $s is [string length $s].
```

In the second example, the Jacl interpreter performs variable and command substitution on the second argument from the **puts** command. In the third command, substitutions are prevented so the string is printed as it is.

Procedures and scope:

Jacl uses the **proc** command to define procedures. The basic syntax to define a procedure is the following:

```
proc  name  arglist  body
```

The first argument is the name of the procedure being defined. The name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each other. The second argument is a list of parameters to the procedures. The third argument is a command, or more typically a group of commands that form the procedure body. Once defined, a Jacl procedure is used just like any of the built-in commands. For example:

```
proc divide {x y} {
set result [expr $x/$y]
puts $result
}
```

Inside the script, this is how to call devide procedure:

```
divide 20 5
```

And it will give the result like below:

```
4
```

It is not really necessary to use the variable c in this example. The procedure body could also written as:

```
return  [expr  sqrt($a  *  $a  +  $b  *  $b)]
```

The return command is optional in this example because the Jacl interpreter returns the value of the last command in the body as the value of the procedure. So, the procedure body could be reduced to:

```
expr  sqrt($a  *  $a  +  $b  *  $b)
```

The result of the procedure is the result returned by the last command in the body. The return command can be used to return a specific value.

There is a single, global scope for procedure names. You can define a procedure inside another procedure, but it is visible everywhere. There is a different name space for variables and procedures therefore you may have a procedure and a variable with the same name without a conflict. Each procedure has a local scope for variables. Variables introduced in the procedures only exist for the duration of the procedure call. After the procedure returns, those variables are undefined. If the same variable name exists in an outer scope, it is unaffected by the use of that variable name inside a procedure. Variables defined outside the procedure are not visible to a procedure, unless the global scope commands are used.

- **global** command - Global scope is the top level scope. This scope is outside of any procedure. You must make variables defined at the global scope accessible to the commands inside procedure by using the **global** command. The syntax for the **global** command is the following:

  ```
  global  varName1  varName2 ...
  ```

Comments

Use the pound character (#) to make comments.

Command line arguments

The Jacl shells pass the command line arguments to the script as the value of the `argv` variable. The number of command line arguments is given by `argc` variable. The name of the program, or script, is not part of `argv` nor is it counted by `argc`. The `argv` variable is a list. Use the **lindex** command to extract items from the argument list, for example:

```
set  first  [lindex  $argv  0]
set  second  [lindex  $argv  1]
```

Strings and pattern matching

String are the basic data item in the Jacl language. There are multiple commands that you can use to manipulate strings. The general syntax of the **string** command is the following:

```
string operation stringvalue otherargs
```

The `operation` argument determines the action of the string. The second argument is a string value. There may be additional arguments depending on the operation.

The following table includes a summary of the **string** command:

Command	Description

string compare str1 str2	Compares strings lexicographically. Returns 0 if equal, -1 if str1 sorts before str2, else1.
string first str1 str2	Returns the index in str2 of the first occurrence of str1, or -1 if str1 is not found.
string index string index	Returns the character at the specified index.
string last str1 str2	Returns the index in str2 of the last occurrence of str1, or -1 if str1 is not found.
string length string	Returns the number of character in string.
string match pattern str	Returns 1 if str matches the pattern, else 0.
string range str i j	Returns the range of characters in str from i to j
string tolower string	Returns string in lower case.
string toupper string	Returns string in upper case.
string trim string ?chars?	Trims the characters in chars from both ends of string. chars defaults to white space.
string trimleft string ?chars?	Trims the characters in chars from the beginning of string. chars defaults to white space.
string trimright string ?chars?	Trims the characters in chars from the end of string. chars defaults to white space.
string wordend str ix	Returns the index in str of the character after the word containing the character at index ix.
string wordstart str ix	Returns the index in str of the first character in the word containing the character at index ix.

The append command

The first argument of the **append** command is a variable name. It concatenates the remaining arguments onto the current value of the named variable. For example:

```
set  foo  z
=> z

append  foo a b c
=> zabc
```

The regexp command

The **regexp** command provides direct access to the regular expression matcher. The syntax is the following:

```
regexp  ?flags?  pattern  string  ?match  sub1  sub2  ...?
```

The return value is `1` if some part of the string matches the pattern. Otherwise, the return value will be `0`. The pattern does not have to match the whole string. If you need more control than this, you can anchor the pattern to the beginning of the string by starting the pattern with ^, or to the end of the string by ending the pattern with dollar sign, `$`. You can force the pattern to match the whole string by using both characters. For example:

```
set  text1  "This is the first string"
=> This is the first string

regexp  "first string" $text1
=> 1

regexp "second string"  $text1
=> 0
```

Jacl data structures

The basic data structure in the Jacl language is a string. There are two higher level data structures: lists and arrays. Lists are implemented as strings and the structure is defined by the syntax of the string. The syntax rules are the same as for commands. Commands are a particular instance of lists. Arrays are variables that have an index. The index is a string value so you can think of arrays as maps from one string (the index) to another string (the value of the array element).

Jacl lists

The lists of the Jacl language are strings with a special interpretation. In the Jacl language, a list has the same structure as a command. A list is a string with list elements separated by white space. You can use braces or quotes to group together words with white space into a single list element.

The following table includes commands that are related to lists:

Command	Description
list arg1 arg2	Creates a list out of all its arguments.
lindex list i	Returns the i'th element from list.
llength list	Returns the number of elements in list.
lrange list i j	Returns the i'th through j'th elements from list.
lappend listVar arg arg ...	Appends elements to the value of listVar
linsert list index arg arg ...	Inserts elements into list before the element at position index. Returns a new list.
lreplace list i j arg arg ...	Replaces elements i through j of list with the args. Return a new list.
lsearch mode list value	Returns the index of the element in list that matches the value according to the mode, which is -exact, -glob, or -regexp, -glob is the default. Return -1 if not found.
lsort switches list	Sorts elements of the list according to the switches: -ascii, -integer, -real, -increasing, -decreasing, -command command. Return a new list.
concat arg arg arg ...	Joins multiple lists together into one list.
join list joinString	Merges the elements of a list together by separating them with joinString.
split string splitChars	Splits a string up into list elements, using the characters in splitChars as boundaries between list elements.

Arrays

Arrays are the other primary data structure in the Jacl language. An array is a variable with a string-valued index, so you can think of an array as a mapping from strings to strings. Internally an array is implemented with a hash table. The cost of accessing each element is about the same. The index of an array is delimited by parentheses. The index can have any string value, and it can be the result of variable or command substitution. Array elements are defined with the **set** command, for example:

```
set arr(index) value
```

Substitute the dollar sign ($) to obtain the value of an array element, for example:

```
set foo $arr(index)
```

For example:

```
set fruit(best) kiwi
=> kiwi

set fruit(worst) peach
=> peach

set fruit(ok) banana
=> banana

array get fruit
=> ok banana worst peach best kiwi

array exists fruit
=> 1
```

The following table includes array commands:

Command	Description
array exists arr	Returns 1 if arr is an array variable.
array get arr	Returns a list that alternates between an index and the corresponding array value.
array names arr ?pattern?	Return the list of all indices defined for arr, or those that match the string match pattern.
array set arr list	Initializes the array arr from list, which should have the same form as the list returned by get.
array size arr	Returns the number of indices defined for arr.
array startsearch arr	Returns a search token for a search through arr.
array nextelement arr id	Returns the value of the next element in array in the search identified by the token id. Returns an empty string if no more elements remain in the search.
array anymore arr id	Returns 1 if more elements remain in the search.
array donesearch arr id	Ends the search identified by `id`.

Control flow commands

The following looping commands exist:

- `while`
- `foreach`
- `for`

The following are conditional commands:

- `if`
- `switch`

The following is an error handling command:

- `catch`

The following commands fine-tune control flow:

- `break`
- `continue`
- `return`
- `error`

If Then Else

The **if** command is the basic conditional command. It says that if an expression is true, then run the second line of code, otherwise run a different line of code. The second command body (the else clause) is optional. The syntax of the command is the following:

```
if  boolean  then  body1  else  body2
```

The then and else keywords are optional. For example:

```
if {$x == 0} {
 puts stderr "Divide by zero!"
} else {
 set slope [expr $y/$x]
}
```

Switch

Use the **switch** command to branch to one of many commands depending on the value of an expression. You can choose based on pattern matching as well as simple comparisons. Any number of pattern-body pairs can be specified. If multiple patterns match, only the code body of the first matching pattern is evaluated. The general form of the command is the following:

```
switch  flags  value  pat1  body1  pat2  body2  ...
```

You can also group all the pattern-body pairs into one argument:

```
switch  flags  value  {pat1  body1  pat2  body2  ...}
```

There are four possible flags that determines how value is matched.

- `-exact` Matches the value exactly to one of the patterns.
- `-glob` Uses glob-style pattern matching.
- `-regexp` Uses regular expression pattern matching.
- `--` No flag (or end of flags). Useful when value can begin with a dash (-).

For example:

```
switch  -exact  --  $value  {
 foo  {doFoo;  incr  count(foo)}
 bar {doBar;  return  $count(foo)}
 default   {incr  count(other)}
}
```

If the pattern that is associated with the last body is `default`, then the command body is started if no other patterns match. The `default` keyword only works on the last pattern-body pair. If you use the default pattern on an earlier body, it will be treated as a pattern to match the literal string default.

Foreach

The **foreach** command loops over a command body and assigns a loop variable to each of the values in a list. The syntax is the following:

```
foreach loopVar valueList commandBody
```

The first argument is the name of a variable. The command body runs one time for each element in the loop with the loop variable having successive values in the list. For example:

```
set numbers {1 3 5 7 11 13}
foreach num $numbers {
puts $num
}
```

The result from the previous example will be the following output, assuming that only one server exists in the environment. If there is more than one server, the information for all servers returns:

```
1
3
5
7
11
13
```

While

The **while** command takes two arguments; a test and a command body, for example:

```
while  booleanExpr  body
```

The **while** command repeatedly tests the boolean expression and runs the body if the expression is true (non-zero). For example:

```
set i 0
while {$i < 5} {
puts "i is $i"
incr i}
```

The result from the previous example will be like the following output, assuming that there is only one server. If there is more then one servers, it will print all of the servers:

```
i is 0
i is 1
i is 2
i is 3
i is 4
```

For

The **for** command is similar to the C language for statement. It takes four arguments, for example:

```
for initial test final body
```

The first argument is a command to initialize the loop. The second argument is a boolean expression which determines if the loop body will run. The third argument is a command that runs after the loop body: For example:

```
set numbers {1 3 5 7 11 13}
for {set i 0} {$i < [llength $numbers]} {incr i 1} {
puts "i is $i"
}
```

The result from previous example will be like the following output, assuming that there is only one server in the environment. If there is more then one server, it will print all of the server names:

```
i is 1
i is 3
i is 5
i is 7
i is 11
i is 13
```

Break and continue

You can control loop execution with the **break** and **continue** commands. The **break** command causes an immediate exit from a loop. The **continue** command causes the loop to continue with the next iteration.

Catch

An error will occur if you call a command with the wrong number of arguments or if the command detects some error condition particular to its implementation. An uncaught error prevents a script from running. Use the **catch** command trap such errors. The catch command takes two arguments, for example:

```
catch command ?resultVar?
```

The first argument is a command body. The second argument is the name of a variable that will contain the result of the command or an error message if the command raises an error. The **catch** command returns a value of zero if no error was caught or a value of one if the command catches an error. For example:

```
catch {expr 20 / 5} result
==> 0
puts $result
==> 4
catch {expr text / 5} result
==> 1
puts $result
==> syntax error in expression "text / 5"
```

Return

Use the **return** command to return a value before the end of the procedure body or if a contrast value needs to be returned.

Namespaces

Jacl keeps track of named entities such as variables, in namespaces. The wsadmin tool also adds entries to the global namespace for the scripting objects, such as, the AdminApp object

When you run a proc command, a local namespace is created and initialized with the names and the values of the parameters in the proc command. Variables are held in the local namespace while you run the proc command. When you stop the proc command, the local namespace is erased. The local namespace of the proc command implements the semantics of the automatic variables in languages such as C and Java.

While variables in the global namespace are visible to the top level code, they are not visible by default from within a proc command. To make them visible, declare the variables globally using the **global** command. For the variable names that you provide, the global command creates entries in the local namespace that point to the global namespace entries that actually define the variables.

If you use a scripting object provided by the wsadmin tool in a proc, you must declare it globally before you can use it, for example:

```
proc { ... } {
  global AdminConfig
  ... [$AdminConfig ...]
}
```

Calling scripts using another script

Use the **source** command to call a Jacl script from another Jacl script. For example:

Create a script called *test1.jacl*.

Create a script called *testProcedure.jacl*.

```
proc printName {first last} {
 puts "My name is $first $last"
}
```

Pass the following path as a script argument.

You must use forward slashes (/) as your path separator. Backward slashes (\) will not work.

Redirection using the exec command

The following Jacl **exec** command for redirection does not work on Linux® platforms:

```
eval exec ls -l > /tmp/out
```

The **exec** command of the Jacl scripting language does not fully support redirection therefore it might cause problems on some platforms.

Do not use redirection when using the **exec** command of the Jacl language. Instead, you can save the **exec** command for redirection in a variable and write it to a file, for example:

```
open /tmp/out w puts $fileId $result close $fileId
```

In some cases, you can also perform a redirection using shell and a .sh command redirection, not a redirection issued by Tcl.

Jython

Jython is an alternate implementation of Python, and is written entirely in Java.

The wsadmin tool uses Jython V2.1. The following information is a basic summary of the Jython syntax. In all sample code, the => notation at the beginning of a line represents command or function output.

Note: The product uses a Jython version that does not support Microsoft® Windows® 2003 or Windows Vista operating systems.

Basic function

The function is either the name of a built-in function or a Jython function. For example, the following functions return ″Hello, World!″ as the output:

```
print "Hello, World!"
=>Hello, World!

import sys
sys.stdout.write("Hello World!\n")
=>Hello, World!
```

In the example, `print` identifies the standard output stream. You can use the built-in module by running import statements such as the previous example. The statement import runs the code in a module as part of the importing and returns the module object. `sys` is a built-in module of the Python language. In the Python language, modules are name spaces which are places where names are created. Names that reside in modules are called attributes. Modules correspond to files and the Python language creates a module object to contain all the names defined in the file. In other words, modules are name spaces.

Variable

To assign objects to names, the target of an assignment should be on the left side of an equal sign (=) and the object that you are assigning on the right side. The target on the left side can be a name or object component, and the object on the right side can be an arbitrary expression that computes an object. The following rules exist for assigning objects to names:

- Assignments create object references.
- Names are created when you assign them.

- You must assign a name before referencing it.

Variable name rules are similar to the rules for the C language, for example:

- An underscore character (_) or a letter plus any number of letters, digits or underscores

The following reserved words can not be used for variable names:

```
and  assert  break  class  continue
def  del  elif  else  except
exec  finally  for  from  global
if  import  in  is  lambda
not  or  pass  print  raise
return  try  while
```

For example:

```
a  = 5
print a
=> 5

b =  a
print b
=> 5

text1, text2, text3, text4  = 'good', 'bad', 'pretty', 'ugly'
print text3
=> pretty
```

The second example assigns the value of variable a to variable b.

Types and operators

The following list contains a few of the built-in object types:

- Numbers. For example:

  ```
  8, 3.133,  999L,  3+4j

  num1 = int(10)
  print num1
  => 10
  ```

- Strings. For example:

  ```
  'name',  "name's", ''

  print str(12345)
  => '12345'
  ```

- Lists. For example:

  ```
  x = [1, [2,  'free'], 5]
  y = [0, 1, 2, 3]
  y.append(5)
  print y
  => [0, 1, 2, 3, 5]

  y.reverse()
  print y
  => [5, 3, 2, 1, 0]

  y.sort()
  print y
  => [0, 1, 2, 3, 5]

  print list("apple")
  => ['a', 'p', 'p', 'l', 'e']

  print list((1,2,3,4,5))
  => [1, 2, 3, 4, 5]
  ```

```
test = "This is a test"
test.index("test")
=> 10

test.index('s')
=> 3
```

The following list contains a few of the operators:

- x or y

  y is evaluated only if x is false. For example:

  ```
  print 0 or 1
  => 1
  ```

- x and y

  y is evaluated only if x is true. For example:

  ```
  print 0 and 1
  =>  0
  ```

- x + y , x - y

  Addition and concatenation, subtraction. For example:

  ```
  print  6 + 7
  => 13

  text1 = 'Something'
  text2 = ' else'
  print text1 + text2
  => Something else

  list1 = [0, 1, 2, 3]
  list2 = [4, 5, 6, 7]
  print list1 + list2
  => [0, 1, 2, 3, 4, 5, 6, 7]

  print  10 - 5
  => 5
  ```

- x * y, x / y, x % y

  Multiplication and repetition, division, remainder and format. For example:

  ```
  print 5 * 6
  => 30

  print 'test' * 3
  => test test test

  print 30 / 6
  => 5

  print 32 % 6
  => 2
  ```

- x[i], x[i:j], x(...)

  Indexing, slicing, function calls. For example:

  ```
  test = "This is a test"
  print  test[3]
  => s

  print test[3:10]
  => s is a

  print test[5:]
  => is a test
  ```

```
print test[:-4]
=> This is a print len(test)
=> 14
```

- <, <=, >, >=, ==, <>, !=, is is not

  Comparison operators, identity tests. For example:

```
L1 = [1, ('a', 3)]
L2 = [1, ('a', 2)]
L1 < L2, L1 == L2, L1 > L2, L1 <> L2, L1 != L2, L1 is L2, L1 is not L2
=> (0, 0, 1, 1, 1, 0, 1)
```

Backslash substitution

If a statement needs to span multiple lines, you can also add a back slash (\) at the end of the previous line to indicate you are continuing on the next line. Do not use white space characters, specifically tabs or spaces, after the back slash character. For example:

```
text =  "This is a test of a long lines" \
" continuing lines here."
print text
=> This is a test of a long lines continuing lines here.
```

Functions and scope

Jython uses the `def` statement to define functions. Functions related statements include:

- `def`, `return`

  The `def` statement creates a function `object` and assigns it to a name. The `return` statement sends a result object back to the caller. This is optional, and if it is not present, a function exits so that control flow falls off the end of the function body.

- `global`

  The `global` statement declares module-level variables that are to be assigned. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, list functions in a global statement.

The basic syntax to define a function is the following:

```
def name (arg1, arg2, ... ArgN)
   statements
   return value
```

where *name* is the name of the function being defined. It is followed by an open parenthesis, a close parenthesis and a colon. The arguments inside parenthesis include a list of parameters to the procedures. The next line after the colon is the body of the function. A group of commands that form the body of the function. After you define a Jython function, it is used just like any of the built-in functions. For example:

```
def intersect(seq1, seq2):
   res = []
   try:
      for x in seq1:
         if x in seq2:
            res.append(x)
   except:
      pass
   return res
```

To call the function above, use the following command:

```
s1 = "SPAM"
s2 = "SCAM"
intersect(s1, s2)
```

```
=> [S, A, M]

intersect([1,2,3], (1.4))
=> [1]
```

Comments

Make comments in the Jython language with the pound character (#).

Command line arguments

The Jython shells pass the command line arguments to the script as the value of the sys.argv. In wsadmin Jython, the name of the program, or script, is not part of sys.argv. Unlike wsadmin Jython, Jython standalone takes the script file as the first argument to the script. Since sys.argv is an array, use the index command to extract items from the argument list. For example, test.py takes 3 arguments a, b, and c.

```
wsadmin -f test.py  a  b  c
```

test.py content:

```
import sys
first  =  sys.argv[0]
second = sys.argv[1]
third = sys.argv[2]
arglen = len(sys.argv)
```

Basic statements

There are two looping statements: while and for. The conditional statement is `if`. The error handling statement is `try`. Finally, there are some statements to fine-tune control flow: break, continue and pass. The following is a list of syntax rules in Python:

- Statements run one after another until you say otherwise. Statements normally end at the end of the line they appear on. When statements are too long to fit on a single line you can also add a back sash (\) at the end of the prior line to indicate you are continuing on the next line.
- Block and statement boundaries are detected automatically. There are no braces, or begin or end delimiter, around blocks of code. Instead, the Python language uses the indentation of statements under a header in order to group the statements in a nested block. Block boundaries are detected by line indentation. All statements indented the same distance to the right belong to the same block of code until that block is ended by a line less indented.
- Compound statements = header; ':', indented statements. All compound statements in the Python language follow the same pattern: a header line terminated with a colon, followed by one or more nested statements indented under the header. The indented statements are called a block.
- Spaces and comments are usually ignored. Spaces inside statements and expressions are almost always ignored (except in string constants and indentation), so are comments.

If

The `if` statement selects actions to perform. The `if` statement may contain other statements, including other `if` statements. The `if` statement can be followed by one or more optional `elif` statements and ends with an optional `else` block.

The general format of an if looks like the following:

```
if test1
   statements1
elif test2
   statements2
else
   statements3
```

For example:

```
weather = 'sunny'
if weather == 'sunny':
   print "Nice weather"
elif weather == 'raining':
   print "Bad weather"
else:
   print "Uncertain, don't plan anything"
```

While

The `while` statement consists of a header line with a test expression, a body of one or more indented statements, and an optional `else` statement that runs if control exits the loop without running into a break statement. The `while` statement repeatedly executes a block of indented statements as long as a test at the top keeps evaluating a true value. The general format of an while looks like the following:

```
while test1
   statements1
else
   statements2
```

For example:

```
a = 0; b = 10
while a < b:
   print a
   a = a + 1
```

For

The `for` statement begins with a header line that specifies an assignment target or targets, along with an object you want to step through. The header is followed by a block of indented statements which you want to repeat.

The general format of a for statement looks like the following:

```
for target in object:
   statements
else:
   statements
```

It assigns items in the sequence object to the target, one by one, and runs the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as if it were a cursor stepping through the sequence. For example:

```
sum = 0
for x in [1, 2, 3, 4]:
   sum = sum + x
```

Break, continue, and pass

You can control loops with the `break`, `continue` and `pass` statements. The `break` statement jumps out of the closest enclosing loop (past the entire loop statement). The `continue` statements jumps to the top of the closest enclosing loop (to the header line of the loop), and the `pass` statement is an empty statement placeholder.

Try

A statement will raise an error if it is called with the wrong number of arguments, or if it detects some error condition particular to its implementation. An uncaught error aborts execution of a script. The try statement is used to trap such errors. Python try statements come in two flavors, one that handles exceptions and one that executes finalization code whether exceptions occur or not. The `try`, `except`, `else` statement

starts with a try header line followed by a block of indented statements, then one or more optional except clauses that name exceptions to be caught, and an optional `else` clause at the end. The `try`, `finally` statements starts with a try header line followed by a block of indented statements, then finally clause that always runs on the way out whether an exception occurred while the try block was running or not.

The general format of the try, except, else function looks like the following:

```
try:
   statements
except name:
   statements
except name, data:
   statements
else
   statements
```

For example:

```
try:
   myfunction()
except:
   import sys
   print 'uncaught exception', sys.exc_info()

try:
   myfilereader()
except EOFError:
   break
else:
   process next line here
```

The general format of a try and finally looks like the following:

```
try
   statements
finally
   statements
```

For example:

```
def divide(x, y):
   return x / y

def tester(y):
   try:
      print divide(8, y)
   finally:
      print 'on the way out...'
```

Calling scripts using another script

Use the **execfile** command to call a Jython script from another Jython script. For example:

Create a script called *test1.py* that contains the following:

Create a script called *testFunctions.py* that contains the following:

```
def printName(first, last):
   name = first + ' ' + last
   return name
```

Then pass the following path as a script argument:

You must use forward slashes (/) as your path separator. Backward slashes (\) will not work.

Using the wsadmin scripting objects

The wsadmin tool utilizes a set of management objects which allow you to execute commands and command parameters to configure your environment. Use the AdminConfig, AdminControl, AdminApp, AdminTask, and Help objects to perform administrative tasks.

About this task

Each of the management objects have commands that you can use to perform administrative tasks. To use the scripting objects, specify the scripting object, a command, and command parameters. In the following example, `AdminConfig` is the scripting object, `attributes` is the command, and `ApplicationServer` is the command parameter.

Using Jython:

```
print AdminConfig.attributes('ApplicationServer')
```

Using Jacl:

```
$AdminConfig attributes ApplicationServer
```

Administrative functions within the application server are divided into two categories: functions that work with the configuration of application server installations, and functions that work with the currently running objects on application server installations. Scripts work with both categories of objects. For example, an application server is divided into two distinct entities. One entity represents the configuration of the server that resides persistently in a repository on permanent storage.

- Use the **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** to handle configuration functionality.

  The **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** are used when you are managing the configuration of the server that resides persistently in a repository on permanent storage. Use these objects to create, query, change, or remove this configuration without starting an application server process. To use the **AdminTask object**, you must be connected to a running server.
- Use the **AdminControl object** to manage running objects on application server installations.

  The **AdminControl object** is used when managing the running instance of an application server by a *Java Management Extensions (JMX) MBean*. This instance can have attributes that you can interrogate and change, and operations that you can invoke. These operational actions that are taken against a running application server do not have an effect on the persistent configuration of the server. The attributes that support manipulation from an MBean differ from the attributes that the corresponding configuration supports. The configuration can include many attributes that you cannot query or set from the running object. The application server scripting support provides functions to locate configuration objects and running objects. The objects in the configuration do not always represent objects that are currently running. The **AdminControl object** manages running objects.
- Use the **Help Object** to obtain information about the AdminConfig, AdminApp, AdminControl, and AdminTask objects, to obtain interface information about running MBeans, and to obtain help for warnings and error messages.

Help object for scripted administration

The Help object provides general help, online information about running MBeans, and help on messages.

Use the Help object to obtain general help for the other objects supplied by the wsadmin tool for scripting: the AdminApp, AdminConfig, AdminTask, and AdminControl objects. For example, using Jacl, `$Help AdminApp` or using Jython, `Help.Adminapp()`, provides information about the AdminApp object and the available commands.

The Help object also to provides interface information about MBeans running in the system. The commands that you use to get online information about the running MBeans include: **all**, **attributes**, **classname**, **constructors**, **description**, **notification**, **operations**.

You can also use the Help object to obtain information about messages using the **message** command. The **message** command provides aid to understand the cause of a warning or error message and find a solution for the problem. For example, you receive a `WASX7115E` error when running the AdminApp **install** command to install an application, use the following example:

Using Jacl:

```
$Help message WASX7115E
```

Using Jython:

```
print Help.message('WASX7115E')
```

Example output:

```
Explanation: wsadmin failed to read an ear file when
preparing to copy it to a temporary location for AdminApp
processing.  User action: Examine the wsadmin.traceout
log file to determine the problem; there may be file permission problems.
```

The user action specifies the recommended action to correct the problem. It is important to understand that in some cases the user action may not be able to provide corrective actions to cover all the possible causes of an error. It is an aid to provide you with information to troubleshoot a problem.

To see a list of all available commands for the Help object, see the Commands for the Help object article or you can also use the **Help** command, for example:

Using Jacl:

```
$Help help
```

Using Jython:

```
print Help.help()
```

Using the AdminApp object for scripted administration

Use the AdminApp object to manage applications.

Before you begin

This object communicates with the run time application management object in WebSphere Application Server to make application inquires and changes, for example:

- Installing and uninstalling applications
- Listing applications
- Editing applications or modules

Because applications are part of configuration data, any changes that you make to an application are kept in the configuration session, similar to other configuration data. Be sure to save your application changes so that the data transfers from the configuration session to the master repository.

About this task

With the application already installed, the AdminApp object can update application metadata, map virtual hosts to Web modules, and map servers to modules. You must perform any other changes, such as specifying a library for the application to use or setting session management configuration properties, using the AdminConfig object.

You can run the commands for the AdminApp object in local mode. If a server is running, it is not recommended that you run the scripting client in local mode because any configuration changes that are made in local mode will not be reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

To see a list of all available commands for the AdminApp object:

- See the Commands for the AdminApp object article.
- You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminApp help
  ```

  Using Jython:

  ```
  print AdminApp.help()
  ```

Listing applications with the wsadmin tool

You can list installed applications using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

- Query the configuration and create a list of installed applications, for example:
  - Using Jacl:

    ```
    $AdminApp list
    ```

  - Using Jython:

    ```
    print AdminApp.list()
    ```

  where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`list`	is an AdminApp command

  Example output:

  ```
  DefaultApplication
  SampleApp
  app1serv2
  ```

- Query the configuration and create a list of installed applications on a given target scope, for example:

– Using Jacl:

```
$AdminApp list WebSphere:cell=myCell,node=myNode,server=myServer
```

– Using Jython:

```
print AdminApp.list("WebSphere:cell=myCell,node=myNode,server=myServer")
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`list`	is an AdminApp command
`WebSphere:cell=myCell,node=myNode,server=myServer`	is an optional target scope

Example output:

```
DefaultApplication
PlantsByWebSphere
SamplesGallery
ivtApp
query
```

Editing application configurations with the wsadmin tool

Use the wsadmin tool to configure application settings.

About this task

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Edit the entire application or a single application module. Use one of the following commands:
   - The following command uses the installed application and the command option information to edit the application:
     – Using Jacl:

     ```
     $AdminApp edit appname {options}
     ```

     – Using Jython list:

     ```
     AdminApp.edit('appname', ['options'])
     ```

     – Using Jython string:

     ```
     AdminApp.edit('appname', '[options]')
     ```

     where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`edit`	is an AdminApp command
`appname`	is the name of application or application module to edit. For the application module name, use the module name returned from **listModules** command as the value.
`{options}`	is a list of edit options and tasks similar to the ones for the install command

- The following command changes the application information by prompting you through a series of editing tasks:
  - Using Jacl:

    ```
    $AdminApp editInteractive appname
    ```

  - Using Jython:

    ```
    AdminApp.editInteractive('appname')
    ```

  where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`editInteractive`	is an AdminApp command
appname	is the name of application or application module to edit. For the application module name, use the module name returned from **listModules** command as the value.

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Using the AdminControl object for scripted administration

The AdminControl scripting object is used for operational control. It communicates with MBeans that represent live objects running a WebSphere server process.

Before you begin

It includes commands to query existing running objects and their attributes and invoke operation on the running objects. In addition to the operational commands, the AdminControl object supports commands to query information on the connected server, convenient commands for client tracing, reconnecting to a server, and start and stop server for network deployment environment.

About this task

Many of the operational commands have two sets of signatures so that they can either invoke using string based parameters or using Java Management Extension (JMX) objects as parameters. Depending on the server process to which a scripting client is connected, the number and type of MBeans available varies. When connected to an application server, only MBeans running in that application server are visible.

If a scripting client is connected to a deployment manager, then all MBeans in all server processes are visible. If a scripting client is connected to a node agent, all MBeans in all server processes on that node are accessible.

The following steps provide a general method to manage the cycle of an application:

- Install the application.
- Edit the application.

- Update the application.
- Uninstall the application.

To see a list of all available commands for the AdminControl object:
- See the Commands for the AdminControl object article.
- You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminControl help
  ```

  Using Jython:

  ```
  print AdminControl.help()
  ```

ObjectName, Attribute, and AttributeList classes

WebSphere Application Server scripting commands use the underlying Java Management Extensions (JMX) classes, ObjectName, Attribute, and AttributeList, to manipulate object names, attributes and attribute lists respectively.

The WebSphere Application Server ObjectName class uniquely identifies running objects. The ObjectName class consists of the following elements:
- The domain name `WebSphere`.
- Several key properties, for example:
  - **`type`** - Indicates the type of object that is accessible through the MBean, for example, ApplicationServer, and EJBContainer.
  - **`name`** - Represents the display name of the particular object, for example, MyServer.
  - **`node`** - Represents the name of the node on which the object runs.
  - **`process`** - Represents the name of the server process in which the object runs.
  - **`mbeanIdentifier`** - Correlates the MBean instance with corresponding configuration data.

When ObjectName classes are represented by strings, they have the following pattern:

```
[domainName]:property=value[,property=value]*
```

For example, you can specify `WebSphere:name="My Server",type=ApplicationServer,node=n1,*` to specify an application server named `My Server` on node n1. (The asterisk (*) is a wildcard character, used so that you do not have to specify the entire set of key properties.) The AdminControl commands that take strings as parameters expect strings that look like this example when specifying running objects (MBeans). You can obtain the object name for a running object with the **getObjectName** command.

Attributes of these objects consist of a name and a value. You can extract the name and value with the **getName** and the **getValue** methods that are available in the javax.management.Attribute class. You can also extract a list of attributes.

Example: Collecting arguments for the AdminControl object

This example shows how to use multiple arguments with the AdminControl object.

Verify that the arguments parameter is a single string. Each individual argument in the string can contain spaces. Collect each argument that contains spaces in some way.
- An example of how to obtain an MBean follows:

  Using Jacl:

  ```
  set am [$AdminControl queryNames type=ApplicationManager,process=server1,*]
  ```

  Using Jython:

  ```
  am = AdminControl.queryNames('type=ApplicationManager,process=server1,*')
  ```
- Multiple ways exist to collect arguments that contain spaces. Choose one of the following alternatives:

  Using Jacl:
  - `$AdminControl invoke $am startApplication {"JavaMail Sample"}`
  - `$AdminControl invoke $am startApplication {{JavaMail Sample}}`

- `$AdminControl invoke $am startApplication "\"JavaMail Sample\""`

Using Jython:
- `AdminControl.invoke(am, 'startApplication', '[JavaMail Sample]')`
- `AdminControl.invoke(am, 'startApplication', '\"JavaMail Sample\"')`

Example: Identifying running objects

Use the AdminControl object to interact with running MBeans.

In the WebSphere Application Server, MBeans represent running objects. You can interrogate the MBean server to see the objects it contains.

- Use the **queryNames** command to see running MBean objects. For example:

  Using Jacl:

  ```
  $AdminControl queryNames *
  ```

  Using Jython:

  ```
  print AdminControl.queryNames('*')
  ```

  This command returns a list of all MBean types. Depending on the server to which your scripting client attaches, this list can contain MBeans that run on different servers:
  - If the client attaches to a stand-alone WebSphere Application Server, the list contains MBeans that run on that server.
  - If the client attaches to a node agent, the list contains MBeans that run in the node agent and MBeans that run on all application servers on that node.
  - If the client attaches to a deployment manager, the list contains MBeans that run in the deployment manager, all of the node agents communicating with that deployment manager, and all application servers on the nodes served by those node agents.
- The list that the `queryNames` command returns is a string representation of JMX `ObjectName` objects. For example:

  ```
  WebSphere:cell=MyCell,name=TraceService,mbeanIdentifier=TraceService,
  type=TraceService,node=MyNode,process=server1
  ```

  This example represents a `TraceServer` object that runs in *server1* on *MyNode*.
- The single queryNames argument represents the `ObjectName` object for which you are searching. The asterisk ("*") in the example means return all objects, but it is possible to be more specific. As shown in the example, `ObjectName` has two parts: a domain, and a list of key properties. For MBeans created by the WebSphere Application Server, the domain is WebSphere. If you do not specify a domain when you invoke queryNames, the scripting client assumes the domain is WebSphere. This means that the first example query above is equivalent to:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:*
  ```

  Using Jython:

  ```
  AdminControl.queryNames('WebSphere:*')
  ```
- WebSphere Application Server includes the following key properties for the `ObjectName` object:
  - name
  - type
  - cell
  - node
  - process
  - mbeanIdentifier

  These key properties are common. There are other key properties that exist. You can use any of these key properties to narrow the scope of the **queryNames** command. For example:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Server,node=myNode,*
  ```

  Using Jython:

```
AdminControl.queryNames('WebSphere:type=Server,node=myNode,*')
```

This example returns a list of all MBeans that represent server objects running the node *myNode*. The, * at the end of the `ObjectName` object is a JMX wildcard designation. For example, if you enter the following:

Using Jacl:

```
$AdminControl queryNames WebSphere:type=Server,node=myNode
```

Using Jython:

```
print AdminControl.queryNames('WebSphere:type=Server,node=myNode')
```

you get an empty list back because the argument to queryNames is not a wildcard. There is no Server MBean running that has exactly these key properties and no others.

- If you want to see all the MBeans representing applications running on a particular node, invoke the following example:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Application,node=myNode,*
  ```

  Using Jython:

  ```
  print AdminControl.queryNames('WebSphere:type=Application,node=myNode,*')
  ```

Specifying running objects using the wsadmin tool

Use scripting and the wsadmin tool to specify running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to specify running objects:

1. Obtain the configuration ID with one of the following ways:
   - Obtain the object name with the **completeObjectName** command, for example:
     - Using Jacl:

       ```
       set var [$AdminControl completeObjectName template]
       ```

     - Using Jython:

       ```
       var = AdminControl.completeObjectName(template)
       ```

       where:

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`completeObjectName`	is an AdminControl command
`template`	is a string containing a segment of the object name to be matched. The template has the same format as an object name with the following pattern: `[domainName]:property=value[,property=value]*`. See Object name, Attribute, Attribute list for more information.

If there are several MBeans that match the template, the **completeObjectName** command only retuns the first match. The matching MBean object name is then assigned to a variable.

To look for *server1* MBean in *mynode*, use the following example:

– Using Jacl:

```
set server1 [$AdminControl completeObjectName node=mynode,type=Server,name=server1,*]
```

– Using Jython:

```
server1 = AdminControl.completeObjectName('node=mynode,type=Server,name=server1,*')
```

- Obtain the object name with the **queryNames** command, for example:
  – Using Jacl:

```
set var [$AdminControl queryNames template]
```

  – Using Jython:

```
var = AdminControl.queryNames(template)
```

where:

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application server process.
`queryNames`	is an AdminControl command
`template`	is a string containing a segment of the object name to be matched. The template has the same format as an object name with the following pattern: `[domainName]:property=value[,property=value]*`

2. If there are more than one running objects returned from the **queryNames** command, the objects are returned in a list syntax. One simple way to retrieve a single element from the list is to use the **lindex** command in Jacl and **split** command in Jython. The following example retrieves the first running object from the server list:
   - Using Jacl:

```
set allServers [$AdminControl queryNames type=Server,*]
set aServer [lindex $allServers 0]
```

   - Using Jython:

```
allServers = AdminControl.queryNames('type=Server,*')

# get line separator
import  java
lineSeparator = java.lang.System.getProperty('line.separator')

aServer = allServers.split(lineSeparator)[0]
```

   For other ways to manipulate the list and then perform pattern matching to look for a specified configuration object, refer to the Jacl syntax.

Results

You can now use the running object in with other AdminControl commands that require an object name as a parameter.

Identifying attributes and operations for running objects with the wsadmin tool

You can use scripting to identify attributes and operations for running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Use the **attributes** or **operations** commands of the Help object to find information on a running MBean in the server.

1. Specify a running object.
2. Use the **attributes** command to display the attributes of the running object:
   - Using Jacl:
     ```
     $Help attributes MBeanObjectName
     ```
   - Using Jython:
     ```
     Help.attributes(MBeanObjectName)
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`Help`	is the object that provides general help and information for running MBeans in the connected server process
`attributes`	is a Help command
`MBeanObjectName`	is the string representation of the MBean object name that is obtained in step 2

3. Use the **operations** command to find out the operations that are supported by the MBean:
   - Using Jacl:
     ```
     $Help operations MBeanObjectname
     ```

     or
     ```
     $Help operations MBeanObjectname operationName
     ```
   - Using Jython:
     ```
     Help.operations(MBeanObjectname)
     ```

     or
     ```
     Help.operations(MBeanObjectname, operationName)
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`Help`	is the object that provides general help and information for running MBeans in the connected server process
`operations`	is a Help command
`MBeanObjectname`	is the string representation of the MBean object name that is obtained in step number 2
`operationName`	(optional) is the specified operation from which you want to obtain detailed information

   If you do not provide the operationName value, all the operations that are supported by the MBean return with the signature for each operation. If you specify the operationName value, only the operation

that you specify returns and it contains details which include the input parameters and the return value. To display the operations for the server MBean, use the following example:

- Using Jacl:

```
set server [$AdminControl completeObjectName type=Server,name=server1,*]
$Help operations $server
```

- Using Jython:

```
server = AdminControl.completeObjectName('type=Server,name=server1,*')
print Help.operations(server)
```

To display detailed information about the stop operation, use the following example:

- Using Jacl:

```
$Help operations $server stop
```

- Using Jython:

```
print Help.operations(server, 'stop')
```

Performing operations on running objects using the wsadmin tool

You can use scripting to invoke operations on running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to perform operations on running objects:

1. Obtain the object name of the running object. For example:
   - Using Jacl:

     `$AdminControl completeObjectName` *`name`*
   - Using Jython:

     `AdminControl.completeObjectName(`*`name`*`)`

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
`name`	is a fragment of the object name. It is used to find the matching object name. For example: `type=Server,name=serv1,*`. It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, etc.

2. Set the s1 variable to the running object, for example:
   - Using Jacl:

     `set s1 [$AdminControl completeObjectName type=`*`Server`*`,name=`*`server1`*`,*]`
   - Using Jython:

     `s1 = AdminControl.completeObjectName('type=`*`Server`*`,name=`*`server1`*`,*')`

   where:

`set`	is a Jacl command

s1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminControl	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
completeObjectName	is an AdminControl command
type	is the object name property key
Server	is the name of the object
name	is the object name property key
server1	is the name of the server where the operation is invoked

3. Invoke the operation. For example:
   - Using Jacl:
     ```
     $AdminControl invoke $s1 stop
     ```
   - Using Jython:
     ```
     AdminControl.invoke(s1, 'stop')
     ```

   where:

$	is a Jacl operator for substituting a variable name with its value
AdminControl	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
invoke	is an AdminControl command
s1	is the ID of the server that is specified in step number 3
stop	is an operation to invoke on the server

Example

The following example is for operations that require parameters:

- Using Jacl:
  ```
  set traceServ [$AdminControl completeObjectName type=TraceService,process=server1,*]
  $AdminControl invoke $traceServ appendTraceString "com.ibm.ws.management.*=all=enabled"
  ```
- Using Jython:
  ```
  traceServ = AdminControl.completeObjectName('type=TraceService,process=server1,*')
  AdminControl.invoke(traceServ, 'appendTraceString',  "com.ibm.ws.management.*=all=enabled")
  ```

Modifying attributes on running objects with the wsadmin tool

Use scripting and the wsadmin tool to modify attributes on running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to modify attributes on running objects:

1. Obtain the name of the running object, for example:
   - Using Jacl:

```
$AdminControl completeObjectName name
```

- Using Jython:

```
AdminControl.completeObjectName(name)
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans that run in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
name	is a fragment of the object name that is used to find the matching object name. For example: `type=TraceService,node=mynode,*`. This value can be any valid combination of domain and key properties, for example, type, name, cell, node, process, and so on.

2. Set the ts1 variable to the running object, for example:
   - Using Jacl:

```
set ts1 [$AdminControl completeObjectName name]
```

   - Using Jython:

```
ts1 = AdminControl.completeObjectName(name)
```

   where:

`set`	is a Jacl command
`ts1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
name	is a fragment of the object name. It is used to find the matching object name. For example: `type=TraceService,node=mynode,*`. It can be any valid combination of domain and key properties, for example, type, name, cell, node, process, and so on.

3. Modify the running object, for example:
   - Using Jacl:

```
$AdminControl setAttribute $ts1 ringBufferSize 10
```

   - Using Jython:

```
AdminControl.setAttribute(ts1, 'ringBufferSize', 10)
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`setAttribute`	is an AdminControl command
`ts1`	evaluates to the ID of the server specified in step number 3

ringBufferSize	is an attribute of modify objects
10	is the value of the ringBufferSize attribute

You can also modify multiple attribute name and value pairs, for example:

- Using Jacl:

```
set ts1 [$AdminControl completeObjectName type=TraceService,process=server1,*]
$AdminControl setAttributes $ts1 {{ringBufferSize 10} {traceSpecification com.ibm.*=all=disabled}}
```

- Using Jython list:

```
ts1 = AdminControl.completeObjectName('type=TraceService,process=server1,*')
AdminControl.setAttributes(ts1, [['ringBufferSize', 10], ['traceSpecification',  'com.ibm.*=all=disabled']])
```

- Using Jython string:

```
ts1 =AdminControl.completeObjectName('type=TraceService,process=server1,*')
AdminControl.setAttributes(ts1, '[[ringBufferSize 10] [traceSpecification  com.ibm.*=all=disabled]]')
```

The new attribute values are returned to the command line.

Synchronizing nodes with the wsadmin tool

You can propagate node changes using scripting and the wsadmin tool.

Before you begin

There are two ways to complete this task. This topic uses the AdminControl object to synchronize nodes. Alternatively, you can use the node administration scripts in the AdminNodeManagement script library to synchronize a specific node, or to synchronize all active nodes.

About this task

A node synchronization is necessary in order to propagate configuration changes to the affected node or nodes. By default, this situation occurs periodically, as long as the node can communicate with the deployment manager. You can propagate changes explicitly by performing the following steps:

1. Set the variable for node synchronization.
   - Using Jacl:

     ```
     set Sync1 [$AdminControl completeObjectName type=NodeSync,node=myNodeName,*]
     ```

   - Using Jython:

     ```
     Sync1 = AdminControl.completeObjectName('type=NodeSync,node=myNodeName,*')
     ```

   where:

set	is a Jacl command
Sync1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminControl	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
completeObjectName	is an AdminControl command
type=NodeSync,node=*myNodeName*	is a fragment of the object name. The complete name is returned by this command. This fragment is used to find the matching object name which is the SyncNode object for the *myNodeName* node, where *myNodeName* is the name of the node that you use to synchronize configuration changes. For example: type=Server, name=serv1. It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, and so on.

Example output:

```
WebSphere:platform=common,cell=myNetwork,version=5.0,name=node
Sync,mbeanIdentifier=nodeSync,type=NodeSync,node=myBaseNode,
process=nodeagent
```

2. Synchronize the node by issuing the following command:
   - Using Jacl:

     ```
     $AdminControl invoke $Sync1 sync
     ```

   - Using Jython:

     ```
     AdminControl.invoke(Sync1, 'sync')
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans that run in a WebSphere Application Server process
`invoke`	is an AdminControl command
`Sync1`	evaluates the ID of the server that is specified in step number 1
`sync`	is an attribute of modify command

Example output:

```
true
```

You receive an output value of `true`, if the synchronization completes.

Results

Using the AdminConfig object for scripted administration

Use the AdminConfig object to manage the configuration information that is stored in the repository.

Before you begin

This object communicates with the WebSphere Application Server configuration service component to make configuration inquires and changes. You can use it to query existing configuration objects, create configuration objects, modify existing objects, remove configuration objects, and obtain help.

Updates to the configuration through a scripting client are kept in a private temporary area called a workspace and are not copied to the master configuration repository until you run a **save** command. The workspace is a temporary repository of configuration information that administrative clients including the administrative console use. The workspace is kept in the `wstemp` subdirectory of your WebSphere Application Server installation. The use of the workspace allows multiple clients to access the master configuration. If the same update is made by more than one client, it is possible that updates made by a scripting client will not save because there is a conflict. If this occurs, the updates will not be saved in the configuration unless you change the default save policy with the **setSaveMode** command.

About this task

The AdminConfig commands are available in both connected and local modes. If a server is currently running, it is not recommended that you run the scripting client in local mode because the configuration changes made in the local mode is not reflected in the running server configuration and vice versa. In connected mode, the availability of the AdminConfig commands depend on the type of server to which a scripting client is connected in a Network Deployment installation.

The AdminConfig commands are available only if a scripting client is connected to a deployment manager. When connected to a node agent or an application server, the AdminConfig commands will not be available because the configuration for these server processes are copies of the master configuration that resides in the deployment manager. The copies are created in a node machine when configuration synchronization occurs between the deployment manager and the node agent. You should make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

- The following steps provide a general method to update a configuration object:
  1. Identify the configuration type and the corresponding attributes.
  2. Query an existing configuration object to obtain a configuration ID to use.
  3. Modify the existing configuration object or create a one.
  4. Save the configuration.
- See the Commands for the AdminConfig object article. You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminConfig help
  ```

  Using Jython:

  ```
  print AdminConfig.help()
  ```

Creating configuration objects using the wsadmin tool

You can use scripting and the wsadmin tool to create configuration objects.

About this task

Perform this task if you want to create an object. To create new objects from the default template, use the **create** command. Alternatively, you can create objects using an existing object as a template with the **createUsingTemplate** command. You can only use the **createUsingTemplate** command for creation of a server with `APPLICATION_SERVER` type. If you want to create a server with a type other than `APPLICATION_SERVER`, use the **createGenericServer** or the **createWebServer** command.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Use the AdminConfig object **listTemplates** command to list available templates:
   - Using Jacl:

     ```
     $AdminConfig listTemplates JDBCProvider
     ```

   - Using Jython:

     ```
     AdminConfig.listTemplates('JDBCProvider')
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`listTemplates`	is an AdminConfig command
`JDBCProvider`	is an object type

3. Assign the ID string that identifies the existing object to which the new object is added. You can add the new object under any valid object type. The following example uses a node as the valid object type:
   - Using Jacl:

     ```
     set n1 [$AdminConfig getid /Node:mynode/]
     ```

- Using Jython:

```
n1 =  AdminConfig.getid('/Node:mynode/')
```

where:

`set`	is a Jacl command
`$`	is a Jacl operator for substituting a variable name with its value
`n1`	is a variable name
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is an object type
`mynode`	is the host name of the node where the new object is added

4. Specify the template that you want to use:
   - Using Jacl:
   - Using Jython:

   where:

`set`	is a Jacl command
`$`	is a Jacl operator for substituting a variable name with its value
`t1`	is a variable name
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`listTemplates`	is an AdminConfig command
`JDBCProvider`	is an object type
`DB2® JDBC Provider (XA)`	is the name of the template to use for the new object

   If you supply a string after the name of a type, you get back a list of templates with display names that contain the string you supplied. In this example, the AdminConfig **listTemplates** command returns the JDBCProvider template whose name matches *DB2 JDBC Provider (XA)*. This example assumes that the variable that you specify here only holds one template configuration ID. If the environment contains multiple templates with the same string, for example, *DB2 JDBC Provider (XA)*, the variable will hold the configuration IDs of all of the templates. Be sure to identify the specific template that you want to use before you perform the next step, creating an object using a template.
5. Create the object with the following command:
   - Using Jacl:

```
$AdminConfig createUsingTemplate JDBCProvider $n1 {{name newdriver}} $tl
```

   - Using Jython:

```
AdminConfig.createUsingTemplate('JDBCProvider', n1, [['name', 'newdriver']], t1)
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`createUsingTemplate`	is an AdminConfig command

JDBCProvider	is an object type
n1	evaluates the ID of the host node that is specified in step number 3
name	is an attribute of JDBCProvider objects
newdriver	is the value of the name attribute
t1	evaluates the ID of the template that is specified in step number 4

All **create** commands use a template unless there are no templates to use. If a default template exists, the command creates the object.

6. Save the configuration changes.
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Interpreting the output of the AdminConfig attributes command using scripting

Use scripting to interpret the output of the AdminConfig attributes command.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

The **attributes** command is a wsadmin tool on-line help feature. When you issue the **attributes** command, the information that displays does not represent a particular configuration object. It represents information about configuration object types, or object metadata. This article discusses how to interpret the attribute type display.

- Simple attributes

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType1
  "attr1 String"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType1')
  attr1 String
  ```

  Types do not display as fully qualified names. For example, `String` is used for `java.lang.String`. There are no ambiguous type names in the model. For example, `x.y.ztype` and `a.b.ztype`. Using only the final portion of the name is possible, and it makes the output easier to read.

- Multiple attributes

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType2
  "attr1 String" "attr2 Boolean" "attr3 Integer"
  ```

  Using Jython:

```
print AdminConfig.attributes('ExampleType2')
attr1 String attr2 Boolean attr3 Integer
```

All input and output for the scripting client takes place with strings, but `attr2 Boolean` indicates that `true` or `false` are appropriate values. The `attr3 Integer` indicates that string representations of integers (″42″) are needed. Some attributes have string values that can take only one of a small number of predefined values. The wsadmin tool distinguishes these values in the output by the special type name ENUM, for example:

Using Jacl:

```
$AdminConfig attributes ExampleType3
"attr4 ENUM(ALL, SOME, NONE)"
```

Using Jython:

```
print AdminConfig.attributes('ExampleType3')
attr4 ENUM(ALL, SOME, NONE)
```

where: `attr4` is an ENUM type. When you query or set the attribute, one of the values is `ALL`, `SOME`, or `NONE`. The value `A_FEW` results in an error.

- Nested attributes

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType4
  "attr5 String" "ex5 ExampleType5"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType4')
  attr5 String ex5 ExampleType5
  ```

  The `ExampleType4` object has two attributes: a string, and an `ExampleType5` object. If you do not know what is contained in the ExampleType5 object, you can use another **attributes** command to find out. The **attributes** command displays only the attributes that the type contains directly. It does not recursively display the attributes of nested types.

- Attributes that represent lists

  The values of these attributes are object lists of different types. The `*` character distinguishes these attributes, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType5
  "ex6 ExampleType6*"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType5')
  ex6 ExampleType6*
  ```

  In this example, objects of the `ExampleType5` type contain a single attribute, `ex6`. The value of this attribute is a list of `ExampleType6` type objects.

- Reference attributes

  An attribute value that references another object. You cannot change these references using modify commands, but these references display because they are part of the complete representation of the type. Distinguish reference attributes using the `@` sign, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType6
  "attr7 Boolean" "ex7 ExampleType7@"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType6')
  attr7 Boolean ex7 ExampleType7@
  ```

  `ExampleType6` objects contain references to `ExampleType7` type objects.

- Generic attributes

These attributes have generic types. The values of these attributes are not necessarily this generic type. These attributes can take values of several different specific types. When you use the AdminConfig attributes command to display the attributes of this object, the various possibilities for specific types are shown in parentheses, for example:

Using Jacl:

```
$AdminConfig attributes ExampleType8
"name String" "beast AnimalType(HorseType, FishType, ButterflyType)"
```

Using Jython:

```
print AdminConfig.attributes('ExampleType8')
name String beast AnimalType(HorseType, FishType, ButterflyType)
```

In this example, the `beast` attribute represents an object of the generic `AnimalType`. This generic type is associated with three specific subtypes. The wsadmin tool gives these subtypes in parentheses after the name of the base type. In any particular instance of `ExampleType8`, the `beast` attribute can have a value of `HorseType`, `FishType`, or `ButterflyType`. When you specify an attribute in this way, using a modify or create command, specify the type of `AnimalType`. If you do not specify the `AnimalType`, a generic `AnimalType` object is assumed (specifying the generic type is possible and legitimate). This is done by specifying `beast:HorseType` instead of `beast`.

Specifying configuration objects using the wsadmin tool

Specify configuration objects with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

To manage an existing configuration object, identify the configuration object and obtain a configuration ID of the object to use for subsequent manipulation.

1. Obtain the configuration ID in one of the following ways:
   - Obtain the ID of the configuration object with the **getid** command, for example:
     - Using Jacl:

       ```
       set var [$AdminConfig getid /type:name/]
       ```

     - Using Jython:

       ```
       var = AdminConfig.getid('/type:name/')
       ```

     where:

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`/type:name/`	is the hierarchical containment path of the configuration object
`type`	is the object type. The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that is displayed in the administrative console.
name	is the optional name of the object

You can specify multiple /type:name/ value pairs in the string, for example, /type:name/type:name/type:name/. If you specify the type in the containment path without the name, include the colon, for example, /type:/. The containment path must be a path that contains the correct hierarchical order. For example, if you specify /Server:server1/Node:node/ as the containment path, you do not receive a valid configuration ID because Node is a parent of Server and comes before Server in the hierarchy.

This command returns all the configuration IDs that match the representation of the containment and assigns them to a variable.

To look for all the server configuration IDs that reside in the mynode node, use the code in the following example:

– Using Jacl:

```
set nodeServers [$AdminConfig getid /Node:mynode/Server:/]
```

– Using Jython:

```
nodeServers = AdminConfig.getid('/Node:mynode/Server:/')
```

To look for the server1 configuration ID that resides in mynode, use the code in the following example:

– Using Jacl:

```
set server1 [$AdminConfig getid /Node:mynode/Server:server1/]
```

– Using Jython:

```
server1 = AdminConfig.getid('/Node:mynode/Server:server1/')
```

To look for all the server configuration IDs, use the code in the following example:

– Using Jacl:

```
set servers [$AdminConfig getid /Server:/]
```

– Using Jython:

```
servers = AdminConfig.getid('/Server:/')
```

- Obtain the ID of the configuration object with the **list** command, for example:

  – Using Jacl:

  ```
  set var [$AdminConfig list type]
  ```

  or

  ```
  set var [$AdminConfig list type scopeId]
  ```

  – Using Jython:

  ```
  var = AdminConfig.list('type')
  ```

  or

  ```
  var = AdminConfig.list('type', 'scopeId')
  ```

  where:

set	is a Jacl command
var	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
list	is an AdminConfig command
type	is the object type. The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that is displayed in the administrative console.
scopeId	is the configuration ID of a cell, a node, or a server object

This command returns a list of configuration object IDs of a given type. If you specify the *scopeId* value, the list of objects is returned within the specified scope. The returned list is assigned to a variable.

To look for all the server configuration IDs, use the following example:

– Using Jacl:

```
set servers [$AdminConfig list Server]
```

– Using Jython:

```
servers = AdminConfig.list('Server')
```

To look for all the server configuration IDs in the *mynode* node, use the code in the following example:

– Using Jacl:

```
set scopeid [$AdminConfig getid /Node:mynode/]
set nodeServers [$AdminConfig list Server $scopeid]
```

– Using Jython:

```
scopeid = AdminConfig.getid('/Node:mynode/')
nodeServers = AdminConfig.list('Server', scopeid)
```

2. If more than one configuration ID is returned from the **getid** or the **list** command, the IDs are returned in a list syntax. One way to retrieve a single element from the list is to use the **lindex** command. The following example retrieves the first configuration ID from the server object list:

   • Using Jacl:

   ```
   set allServers [$AdminConfig getid /Server:/]
   set aServer [lindex $allServers 0]
   ```

   • Using Jython:

   ```
   allServers = AdminConfig.getid('/Server:/')

    # get line separator
    import  java
    lineSeparator = java.lang.System.getProperty('line.separator')

    arrayAllServers = allServers.split(lineSeparator)
    aServer = arrayAllServers[0]
   ```

   For other ways to manipulate the list and perform pattern matching to look for a specified configuration object, refer to the Jacl syntax.

Results

You can now use the configuration ID in any subsequent AdminConfig commands that require a configuration ID as a parameter.

Listing attributes of configuration objects using the wsadmin tool

You can use scripting to generate a list of attributes of configuration objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to create a list of attributes of configuration objects:

1. List the attributes of a given configuration object type, using the **attributes** command, for example:

   • Using Jacl:

   ```
   $AdminConfig attributes type
   ```

   • Using Jython:

```
AdminConfig.attributes('type')
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`attributes`	is an AdminConfig command
`type`	is an object type

This command returns a list of attributes and its data type.

To get a list of attributes for the JDBCProvider type, use the following example command:

- Using Jacl:

```
$AdminConfig attributes JDBCProvider
```

- Using Jython:

```
AdminConfig.attributes('JDBCProvider')
```

2. List the required attributes of a given configuration object type, using the **required** command, for example:
   - Using Jacl:

```
$AdminConfig required type
```

   - Using Jython:

```
AdminConfig.required('type')
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`required`	is an AdminConfig command
`type`	is an object type

   This command returns a list of required attributes.

   To get a list of required attributes for the JDBCProvider type, use the following example command:

   - Using Jacl:

```
$AdminConfig required JDBCProvider
```

   - Using Jython:

```
AdminConfig.required('JDBCProvider')
```

3. List attributes with defaults of a given configuration object type, using the **defaults** command, for example:
   - Using Jacl:

```
$AdminConfig defaults type
```

   - Using Jython:

```
AdminConfig.defaults('type')
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration

`defaults`	is an AdminConfig command
`type`	is an object type

This command returns a list of all the attributes, types, and defaults.

To get a list of attributes with the defaults displayed for the JDBCProvider type, use the following example command:

- Using Jacl:

  ```
  $AdminConfig defaults JDBCProvider
  ```

- Using Jython:

  ```
  AdminConfig.defaults('JDBCProvider')
  ```

Modifying configuration objects with the wsadmin tool

You can modify configuration objects using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

When using the **modify** command for the AdminConfig object, use the configuration object ID to modify the attribute you want to change. If you use the parent object ID to modify the attribute, the command resets all other attributes that are not specified to the default values. For example, you use the **modify** command to change the monitoring policy settings through its parent object, the process definition object. All attributes for the process definition object that were not modified with the command, such as the **pingInterval** and **pingTimeout** attributes, are reset to their default values.

Perform the following steps to modify a configuration object:

1. Retrieve the configuration ID of the objects that you want to modify, for example:
   - Using Jacl:

     ```
     set jdbcProvider1 [$AdminConfig getid /JDBCProvider:myJdbcProvider/]
     ```

   - Using Jython:

     ```
     jdbcProvider1 = AdminConfig.getid('/JDBCProvider:myJdbcProvider/')
     ```

   where:

`set`	is a Jacl command
`jdbcProvider1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`/JDBCProvider:myJdbcProvider/`	is the hierarchical containment path of the configuration object
`JDBCProvider`	is the object type
`myJdbcProvider`	is the optional name of the object

2. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $jdbcProvider1
     ```

- Using Jython:

  ```
  AdminConfig.show(jdbcProvider1)
  ```

  where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`show`	is an AdminConfig command
`jdbcProvider1`	evaluates to the ID of the host node that is specified in step number 1

3. Modify the attributes of the configuration object, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $jdbcProvider1 {{description "This is my new description"}}
     ```

   - Using Jython list:

     ```
     AdminConfig.modify(jdbcProvider1, [['description', "This is my new description"]])
     ```

   - Using Jython string:

     ```
     AdminConfig.modify(jdbcProvider1, '[[description "This is my new description"]]')
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`jdbcProvider1`	evaluates to the ID of the host node that is specified in step number 1
`description`	is an attribute of server objects
`This is my new description`	is the value of the description attribute

   You can also modify several attributes at the same time. For example:
   - Using Jacl:

     ```
     {{name1 val1} {name2 val2} {name3 val3}}
     ```

   - Using Jython list:

     ```
     [['name1', 'val1'], ['name2', 'val2'], ['name3', 'val3']]
     ```

   - Using Jython string:

     ```
     '[[name1 val1] [name2 val2] [name3 val3]]'
     ```

4. List all of the attributes that can be modified:
   - Using Jacl:

     ```
     $AdminConfig attributes JDBCProvider
     ```

   - Using Jython:

     ```
     print AdminConfig.attributes('JDBCProvider')
     ```

   Example output:

   ```
   $AdminConfig attributes JDBCProvider
   "classpath String*"
   "description String"
   "implementationClassName String"
   "name String"
   ```

```
"nativepath String*"
"propertySet J2EEResourcePropertySet"
"providerType String"
"xa boolean"
```

5. Modify an attribute that has a type of list and collection. By default, if you try to modify an attribute that has a type of list and collection, and the attribute has an existing value in the list, it will append the new value to the existing values. An attribute that has a type of list and collection will have a star (*). In the following example, the attribute classpath has an type of list and collection and the value is String. If you want to replace the existing value, you must change the classpath to be an empty list before you modify the new value. For example:
   - Using Jacl:

     ```
     $AdminConfig modify $jdbcProvider1 {{classpath {}}}

     $AdminConfig modify $jdbcProvider1 [list [list classpath /temp/db2j.jar]]
     ```

   - Using Jython list:

     ```
     AdminConfig.modify(jdbcProvider1, [['description', []]])

     AdminConfig.modify(jdbcProvider1, [['description', '/temp/db2j.jar']]
     ```

   - Using Jython string:

     ```
     AdminConfig.modify(jdbcProvider1, '[]')

     AdminConfig.modify(jdbcProvider1, '[[description /temp/db2j.jar]]')
     ```

6. Save the configuration changes.
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Removing configuration objects with the wsadmin tool

Use this task to delete a configuration object from the configuration repository. This action only affects the configuration.

About this task

If a running instance of a configuration object exists when you remove the configuration, the change has no effect on the running instance.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Assign the ID string that identifies the server that you want to remove:

   Using Jacl:

   ```
   set s1 [$AdminConfig getid /Node:mynode/Server:myserver/]
   ```

   Using Jython:

   ```
   s1 = AdminConfig.getid('/Node:mynode/Server:myserver/')
   ```

   where:

set	is a Jacl command
s1	is a variable name

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is an object type
`mynode`	is the host name of the node from which the server is removed
`Server`	is an object type
`myserver`	is the name of the server to remove

3. Remove the configuration object. For example:
   - Using Jacl:

     ```
     $AdminConfig remove $s1
     ```
   - Using Jython:

     ```
     AdminConfig.remove(s1)
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`remove`	is an AdminConfig command
`s1`	evaluates the ID of the server that is specified in step number 2

4. Save the configuration changes.
5. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```
   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Results

The application server configuration no longer contains a specific server object. Running servers are not affected.

Removing the trust association interceptor class using scripting

Use the wsadmin tool to remove the trust association interceptor class.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Use the following example as a Jacl script file and run it with the ″-f″ option:

Using Jacl:

```
set variableName "com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus"
set cellName $env(local.cell)

foreach taiEntry [$AdminConfig list TAInterceptor] {
   set interceptorClass [lindex [$AdminConfig showAttribute $taiEntry interceptorClassName] 0]
   if { [string compare $interceptorClass $variableName] == 0 } {
       puts "found $interceptorClass"
       puts "Removing the TAIntercepter class '$interceptorClass'"
       set tai taiEntry
       #set t [$AdminConfig getid /Cell:$cellName/TAInterceptor:/]
       #$AdminConfig remove $t
       $AdminConfig remove $taiEntry
       puts "'$interceptorClass' is removed."
       break
   }
}

if { ![info exists tai] } {
    puts "The class '$variableName' does not exist."
}

$AdminConfig save
```

Results

Example output:

```
[root@svtaix23] /tmp
==>/usr/6*/A*/profiles/D*/bin/wsadmin.sh -f tai.jacl

WASX7209I: Connected to process "dmgr" on node svtaix23CellManager01 using SOAP connector;
The type of process is: DeploymentManager
found com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus
Removing the TAIntercepter class 'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus'
'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus' is removed.
```

Changing the application server configuration using the wsadmin tool

You can use the wsadmin AdminConfig and AdminApp objects to make changes to the application server configuration.

About this task

The purpose of this article is to illustrate the relationship between the commands that are used to change the configuration and the files that are used to hold configuration data. This discussion assumes that you have a network deployment installation, but the concepts are very similar for a application server installation.

1. Launch the wsadmin scripting tool using the Jython scripting language.

   For this task, connect the wsadmin scripting client to the deployment manager server in a network deployment environment.
2. Set a variable for creating a server:
   - Using Jacl:

     ```
     set n1 [$AdminConfig getid /Node:mynode/]
     ```
   - Using Jython:

     ```
     n1 = AdminConfig.getid('/Node:mynode/')
     ```

where:

`set`	is a Jacl command
`n1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is the object type
mynode	is the name of the object to modify

3. Create a server with the following command:
   - Using Jacl:

     ```
     set serv1 [$AdminConfig create Server $n1 {{name myserv}}]
     ```

   - Using Jython list:

     ```
     serv1 = AdminConfig.create('Server', n1, [['name', 'myserv']])
     ```

   - Using Jython string:

     ```
     serv1 = AdminConfig.create('Server', n1, '[[name myserv]]')
     ```

   where:

`set`	is a Jacl command
`serv1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Server`	is an AdminConfig object
`n1`	evaluates to the ID of the host node that is specified in step number 1
`name`	is an attribute
myserv	is the value of the name attribute

   After this command completes, some new files can be seen in a workspace used by the deployment manager server on behalf of this scripting client. A workspace is a temporary repository of configuration information that administrative clients use. Any changes made to the configuration by an administrative client are first made to this temporary workspace. For scripting, when a **save** command is invoked on the AdminConfig object, these changes are transferred to the real configuration repository. Workspaces are kept in the `wstemp` subdirectory of a WebSphere Application Server installation.
4. Make a configuration change to the server with the following command:
   - Using Jacl:

     ```
     $AdminConfig modify $serv1 {{stateManagement {{initialState STOP}}}}
     ```

   - Using Jython list:

     ```
     AdminConfig.modify(serv1, [['stateManagement', [['initialState', 'STOP']]]])
     ```

   - Using Jython string:

     ```
     AdminConfig.modify(serv1, '[[stateManagement  [[initialState  STOP]]]]')
     ```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`serv1`	evaluates to the ID of the host node that is specified in step number 2
`stateManagement`	is an attribute
`initialState`	is a nested attribute within the stateManagement attribute
`STOP`	is the value of the initialState attribute

This command changes the initial state of the new server. After this command completes, one of the files in the workspace is changed.

5. Save the configuration changes.
6. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Modifying nested attributes with the wsadmin tool

You can modify nested attributes for a configuration object using scripting and the wsadmin tool.

About this task

The attributes for a WebSphere Application Server configuration object are often deeply nested. For example, a JDBCProvider object has an attribute factory, which is a list of the J2EEResourceFactory type objects. These objects can be DataSource objects that contain a connectionPool attribute with a ConnectionPool type that contains a variety of primitive attributes.

1. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** commands from an operating system command prompt.
2. Obtain the configuration ID of the object, for example:

   Using Jacl:

   ```
   set t1 [$AdminConfig getid /DataSource:TechSamp/]
   ```

   Using Jython:

   ```
   t1=AdminConfig.getid('/DataSource:TechSamp/')
   ```

   where:

`set`	is a Jacl command
`t1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration

`getid`	is an AdminConfig command
`DataSource`	is the object type
TechSamp	is the name of the object that will be modified

3. Modify one of the object parents and specify the location of the nested attribute within the parent, for example:

   Using Jacl:

   ```
   $AdminConfig modify $t1 {{connectionPool {{reapTime 2003}}}}
   ```

   Using Jython list:

   ```
   AdminConfig.modify(t1, [["connectionPool", [["reapTime", 2003]]]])
   ```

   Using Jython string:

   ```
   AdminConfig.modify(t1, '[[connectionPool [[reapTime 2003]]]]')
   ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`t1`	evaluates to the configuration ID of the datasource in step number 2
`connectionPool`	is an attribute
`reapTime`	is a nested attribute within the connectionPool attribute
2003	is the value of the reapTime attribute

4. Save the configuration by issuing an AdminConfig **save** command. For example:

   Using Jacl:

   ```
   $AdminConfig save
   ```

   Using Jython:

   ```
   AdminConfig.save()
   ```

   Use the **reset** command of the AdminConfig object to undo changes that you made to your workspace since your last save.

Example

An alternative way to modify nested attributes is to modify the nested attribute directly, for example:

Using Jacl:

```
set techsamp [$AdminConfig getid /DataSource:TechSamp/]
set pool [$AdminConfig showAttribute $techsamp connectionPool]
$AdminConfig modify $pool {{reapTime 2003}}
```

Using Jython list:

```
techsamp=AdminConfig.getid('/DataSource:TechSamp/')
pool=AdminConfig.showAttribute(techsamp,'connectionPool')
AdminConfig.modify(pool,[['reapTime',2003]])
```

Using Jython string:

```
techsamp=AdminConfig.getid('/DataSource:TechSamp/')
pool=AdminConfig.showAttribute(techsamp,'connectionPool')
AdminConfig.modify(pool,'[[reapTime 2003]]')
```

In this example, the first command gets the configuration id of the DataSource, and the second command gets the connectionPool attribute. The third command sets the reapTime attribute on the ConnectionPool object directly.

Saving configuration changes with the wsadmin tool

Use the wsadmin tool and scripting to save configuration changes to the master configuration repository.

About this task

The wsadmin tool uses the workspace to hold configuration changes. You must save your changes to transfer the updates to the master configuration repository. If a scripting process ends and you have not saved your changes, the changes are discarded.

Use the following commands to save the configuration changes:

1. Using Jacl:

   ```
   $AdminConfig save
   ```

2. Using Jython:

   ```
   AdminConfig.save()
   ```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`save`	is an AdminConfig command

If you are using interactive mode with the wsadmin tool, you will be prompted to save your changes before they are discarded.
If you are using the -c option with the wsadmin tool, changes are automatically saved. On a Unix operating system, if you invoke a command that includes a dollar sign character ($) using the wsadmin -c option, the command line attempts to substitute a variable. To avoid this problem, escape the dollar sign character with a backslash character (\). For example: `wsadmin -c "\$AdminConfig save"`.
If a scripting process ends and no save has been performed, any configuration changes made since the last save are discarded. If there are multiple clients (scripts or browser clients) updating the configuration at the same time, it is possible that the changes requested by a script may not be saved. If this happens, you will receive an exception and you must make the updates again. If the save fails, the updates will not be saved to the configuration. If it succeeds, all updates are saved. To avoid save failures, you can invoke the **save** command after every configuration update.
You can use the **reset** command of the AdminConfig object to undo changes that you made to your configuration since your last save.

Using the AdminTask object for scripted administration

Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Before you begin

The administrative commands run simple and complex commands. They provide more user friendly and task-oriented commands. The administrative commands are discovered dynamically when you start a scripting client. The set of available administrative commands depends on the edition of WebSphere Application Server that you installed. You can use the AdminTask object commands to access these commands.

About this task

Administrative commands are grouped based on their function. You can use administrative command groups to find related commands. For example, the administrative commands that are related to server management are grouped into a server management command group. The administrative commands that are related to the security management are grouped into a security management command group. An administrative command can be associated with multiple command groups because it can be useful for multiple areas of system management. Both administrative commands and administrative command groups are uniquely identified by their name.

Two run modes are always available for each administrative command, namely the *batch* and *interactive mode*. When you use an administrative command in interactive mode, you go through a series of steps to collect your input interactively. This process provides users a text-based wizard and a similar user experience to the wizard in the administrative console. You can also use the **help** command to obtain help for any administrative command and the AdminTask object.

The administrative commands do not replace any existing configuration commands or running object management commands but provide a way to access these commands and organize the inputs. The administrative commands can be available in connected or local mode. The set of available administrative commands is determined when you start a scripting client in connected or local mode. If a server is running, it is not recommended that you run the scripting client in local mode because any configuration changes made in local mode are not reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

Use parameter name and parameter value pairs to specify the parameters of a step in any order. You do not have to specify option parameters. This applies to all commands for the AdminTask object. For example:

```
AdminTask.createCluster('[-clusterConfig [-clusterName cluster1 -preferLocal true]]')
```

To determinethe names of the step parameters, use the following command: `AdminTask.help('command_name', 'step_name')`, as the following example demonstrates::

```
AdminTask.help('createCluster', 'clusterConfig')
```

- Read “Invoking an administrative command in batch mode” on page 64 to use administrative commands in batch mode.
- Read “Invoking an administrative command in interactive mode” on page 68 to use administrative commands in interactive mode.
- Read “Obtaining online help using scripting” to learn how to use scripting for online help.

Obtaining online help using scripting

You can select from three levels of online help for administrative commands.

About this task

The top-level help provides general information for the AdminTask object and associated commands. The second-level help provides information about all of the available administrative commands and command groups. The third-level help provides specific help on a command group, a command, or a step. Command

group-specific help provides descriptions for the command group that you specify and the commands that belong to the associated group. Command-specific help provides description for the specified command, and associated parameters and steps. Step-specific help provides a description for the specified step and the associated parameters. For command and step-specific help, required parameters are marked with an asterisk (*) in the help output.

- To obtain general help, use the following examples:

  Using Jacl:

  ```
  $AdminTask help
  ```

  Using Jython:

  ```
  print AdminTask.help()
  ```

  Example output:

  ```
  WASX8001I: The AdminTask object enables the execution of available
             admin commands. AdminTask commands operate in two modes:
             the default mode is one which AdminTask communicates with the
             WebSphere server to accomplish its task. A local mode is also
             available in which no server communication takes place. The local
             mode of operation is invoked by bringing up the scripting client
             using the command line "-conntype NONE" option or setting the
             "com.ibm.ws.scripting.connectiontype=NONE" property in
             wsadmin.properties file.

  The number of admin commands varies and depends on your WebSphere install.
  Use the following help commands to obtain a list of supported commands
  and their parameters:

  help -commands
        list all the admin commands
  help -commandGroups
        list all the admin command groups
  help commandName
        display detailed information for
          the specified command
  help commandName stepName
        display detailed information for
        the specified step belonging to
        the specified command
  help commandGroupName
        display detailed information for
                 the specified command group

  There are various flavors to invoke an admin command:

  commandName
        invokes an admin command that does not require any argument.

  commandName targetObject
        invokes an admin command with the specified target object
        string, for example, the configuration object name of a
        resource adapter. The expected target object varies with
        the admin command invoked. Use help command to get
        information on the target object of an admin command.

  commandName options
        invokes an admin command with the specified option
        strings. This invocation syntax is used to invoke an
                 admin command that does not require a target object. It
                 is also used to enter interactive mode if "-interactive"
                 mode is included in the options string.

  commandName targetObject options
        invokes an admin command with the specified target
        object and options strings. If "-interactive" is
        included in the options string, then interactive mode
  ```

```
      is entered. The target object and options strings vary
      depending on the admin command invoked. Use help
      command to get information on the target
      object and options.
```

- To list the available command groups, use the following examples:

  Using Jacl:

  ```
  $AdminTask help -commandGroups
  ```

  Using Jython:

  ```
  print AdminTask.help('-commandGroups')
  ```

  Example output:

  ```
  WASX8005I: Available admin command groups:

  ClusterConfigCommands - Commands for configuring application
  server clusters and cluster members.
  JCAManagement - A group of admin commands that helps to configure
  Java2 Connector Architecture(J2C) related resources.
  ```

- To list the available commands, use the code in the following examples:

  Using Jacl:

  ```
  $AdminTask help -commands
  ```

  Using Jython:

  ```
  print AdminTask.help('-commands')
  ```

  Example output:

  ```
  WASX8004I: Available administrative commands:

  copyResourceAdapter - copy the specified J2C resource adapter to the specified scope
  createCluster - Creates a new application server cluster.
  createClusterMember - Creates a new member of an application server cluster.
  createJ2CConnectionFactory - Create a J2C connection factory
  deleteCluster - Delete the configuration of an application server cluster.
  deleteClusterMember - Deletes a member from an application server cluster.
  listConnectionFactoryInterfaces - list all of the
  defined connection factory interfaces on the
  specified J2C resource adapter.
  listJ2CConnectionFactories - List J2C connection factories that have a specified
  connection factory interface defined in the specified J2C resouce adapter
  createJ2CAdminObject - Create a J2C administrative object.
  listAdminObjectInterfaces - List all the defined administrative object interfaces
  on the specified J2C resource adapter.
  interface on the specified J2C resource adapter.
  listJ2CAdminObjects - List the J2C administrative objects that have a specified
  administrative object interface defined in the specified J2C resource adapter.
  createJ2CActivationSpec - Create a J2C activation specification.
  listMessageListenerTypes - list all of the defined messageListener
  type on the specified J2C resource adapter.
  listJ2CActivationSpecs - List the J2C activation specifications that have a
  specified message listener type defined in the specified J2C resource adapter.
  ```

- To obtain help about a command group, use the following examples:

  Using Jacl:

  ```
  $AdminTask help JCAManagement
  ```

  Using Jython:

  ```
  print AdminTask.help('JCAManagement')
  ```

  Example output:

  ```
  WASX8007I: Detailed help for command group: JCAManagement

  Description: A group of administrative commands that help to
  configure Java 2 Connector Architecture (J2C)-related resources.

  Commands:
  ```

```
createJ2CConnectionFactory - Create a J2C connection factory
listConnectionFactoryInterfaces - list all of the defined connection
factory interfaces on the specified J2C resource adapter.
listJ2CConnectionFactories - List J2C connection factories that have
a specified connection factory interface defined in the
specified J2C resouce adapter.
createJ2CAdminObject - Create a J2C administrative object.
listAdminObjectInterfaces - List all the defined administrative
object interfaces on the specified J2C resource adapter.
listJ2CAdminObjects - List the J2C administrative objects that have a
specified administrative object interface defined in the
specified J2C resource adapter.
createJ2CActivationSpec - Create a J2C activation specification.
listMessageListenerTypes - list all of the defined
message listener types on the specified J2C resource adapter.
listJ2CActivationSpecs - List the J2C activation specifications that
have a specified message listener type defined in the
specified J2C resource adapter.
copyResourceAdapter - copy the specified J2C resource
adapter to the specified scope.
```

- To obtain help about an administrative command, use the following examples:

  Using Jacl:

  ```
  $AdminTask help createJ2CConnectionFactory
  ```

  Using Jython:

  ```
  print AdminTask.help('createJ2CConnectionFactory')
  ```

  Example output:

  ```
  WASX8006I: Detailed help for command: createJ2CConnectionFactory

  Description: Create a J2C connection factory

  *Target object: The parent J2C resource adapter of the created J2C connection factory.

  Arguments:
  *connectionFactoryInterface - A connection factory interface that is defined in the deployment
    description of the parent J2C resource adapter.
  *name - The name of the J2C connection factory.
  *jndiName - The JNDI name of the created J2C connection factory.
  description - The description for the created J2C connection factory.
  authDataAlias - the authentication data alias of the created J2C connection factory.

  Steps:
  None
  ```

  In the command-specific help output that is previously listed, an administrative command is divided into three input areas: target object, arguments, and steps. Each area can require input depending on the administrative command. If an area requires input, each input is described by its name and a description; except for the target object area, which contains the description of the target object only. When you use an administrative command in batch mode, you can use any input name that resides in the argument area as the argument name.

  If an input is required, an asterisk (*) is located before the name. If an area does not require an input, it is marked `None`. The following example uses the help output for the **createJ2CConnectionFactory** command:

  – The target object area requires the configuration object name of a J2CResourceAdapter.

  – In the arguments area, there are five inputs with three being required inputs. The argument names are connectionFactoryInterface, name, jndiName, description, and authDataAlias. These names are used as the parameter names in the option string to run an administrative command in batch mode, for example:

    ```
    -connectionFactoryInterface javax.resource.cci.ConnectionFactory -name newConnectionFactory
    -jndiName CF/newConnectionFactory
    ```

See “Administrative command invocation syntax” on page 1320 for more information about specifying argument options.

– No step is associated with this administrative command.

- To obtain help on a command step, use the step-specific help.

  Step-specific help provides the following data:

  – A description for the command step.
  – Information indicating if this step supports collection. A collection includes objects of the same type. In a command step, a collection contains objects that have the same set of parameters.
  – Information regarding each step parameter with its name and description. If a step parameter is required, an asterisk (*) is located in front of the name.

  The following example obtains help on a command step:

  Using Jacl:

  ```
  $AdminTask help createCluster clusterConfig
  ```

  Using Jython:

  ```
  print AdminTask.help('createCluster', 'clusterConfig')
  ```

  Example output:

  ```
  WASX8013I: Detailed help for step: clusterConfig

  Description: Specifies the configuration of the new server cluster.

  Collection: No

  Arguments:
    *clusterName - Name of server cluster.
    preferLocal - Enables node-scoped routing optimization for the cluster.
  ```

  This example indicates the following information about the clusterConfig step:

  – This step does not support collection. Only one set of parameter values for the clusterName and perferLocal parameters is supported.
  – This step contains two input arguments with one argument that is indicated as required. The required arguments is clusterName and the non-required parameter is preferLocal. The syntax to provide step parameter values is different from the command argument values. You have to provide all argument values of a step and provide them in the exact order as displayed in the step specific help. For any optional argument that you do not want to specify a value, put double quotes (″″) in place of a value. If a command step is a collection type, for example, it can contain multiple objects where each object has the same set of arguments, you can specify multiple objects with each object enclosed by its own pair of braces. To run an administrative command in batch mode and to include this step in the option string, use the following syntax:

    Using Jacl:

    ```
    -clusterConfig {{newCluster false}}
    ```

    Using Jython:

    ```
    -clusterConfig [[newCluster false]]
    ```

    See “Administrative command invocation syntax” on page 1320 for more information about specifying parameter options.

- Use a wildcard character to search for help for a specific command. You can use a regular Java expression pattern or a wildcard pattern to specify command name for AdminTask.help('–commands') and AdminConfig list, types and listTemplates functions.

  – To use a regular Java expression pattern to search for the administrative command names that start with `create`, specify:

    ```
    print AdminTask.help("-commands", "create.*")
    ```

  – To use a wildcard search pattern to search for the administrative command names that start with `create`, specify:

```
print AdminTask.help("-commands", "create*")
```

– To use a Java expression pattern to search for the administrative command names that contain `SSLConfig`, specify:

```
print AdminTask.help("-commands", ".*SSLConfig.*")
```

– To use a wildcard search pattern to search for the administrative command names that contain `SSLConfig`, specify:

```
print AdminTask.help("-commands", "*SSLConfig*")
```

Invoking an administrative command in batch mode

Use this commands to invoke an administrative command in batch mode.

About this task

To invoke an administrative command in interactive mode, see “Invoking an administrative command in interactive mode” on page 68.

1. Invoke the AdminTask object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Issue one of the following commands:
   - If an administrative command does not have a target object and an argument, use the following command:

     Using Jacl:

     `$AdminTask commandName`

     Using Jython:

     `AdminTask.commandName()`

     where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object allowing administrative command management
commandName	is the name of the administrative command to invoke

   - If an administrative command includes a target object but does not include any arguments or steps, use the following command:

     Using Jacl:

     `$AdminTask commandName targetObject`

     Using Jython:

     `AdminTask.commandName(targetObject)`

     where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
targetObject	is the target object string for the invoked administrative command. The expect target object varies with each administrative command. View the online help for the invoked administrative command to learn more about what you should specify as the target object.

- If an administrative command includes an argument or a step but does not include a target object, use the following command:

  Using Jacl:

  `$AdminTask commandName options`

  Using Jython:

  `AdminTask.commandName(options)`

  where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke

options	is the option string for the invoked administrative command. Depending on which administrative command you are invoking, the administrative command can have required or optional option values. The options string is different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps listed on the online administrative command help are specified as options in the option string.
	
	Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. If the invoked administrative command includes target objects, arguments, or steps, then the –interactive option is available to enter interactive mode. For example, using the output of the following online help for the listDataSource command:
	
	`WASX8006I: Detailed help for command: exportServer`
	
	`Description: export the configuration of a server to a config archive.`
	
	`Target object: None`
	
	`Arguments:`
	`*serverName - the name of a server`
	`*nodeName - the name of a node. This parameter becomes optional if the specified server name is unique across the cell.`
	`*archive - the fully qualified file path of a config archive.`
	
	`Steps:`
	`None`
	
	Option names are specified with a dash before the names. Three options are required for this administrative command. The required options are -serverName, -nodename, and -archive. In addition, the -interactive option is available. Options are specified in the option string, which is enclosed by a pair of braces ({}) in Jacl and a pair of brackets ([]) in Jython.

- If an administrative command includes a target object, and arguments or steps:

  Using Jacl:

```
$AdminTask commandName targetObject options
```

Using Jython:

```
AdminTask.commandName(targetObject, options)
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke

targetObject	is the target object string for the invoked administrative command. The expected target object varies with each administrative command. View the online help for the invoked administrative command to obtain information about what to specify as a target object. For example, using the output of the following online help for createJ2CConnectionFactory:
	`WASX8006I: Detailed help for command: createJ2CConnectionFactory`
	
	`Description: Create a J2C connection factory`
	
	`*Target object: The parent J2C resource adapter of the created J2C connection factory.`
	
	`Arguments:`
	`*connectionFactoryInterface - A connection factory interface that is defined in the deployment description of the parent J2C resource adapter.`
	`*name - The name of the J2C connection factory.`
	`*jndiName - The JNDI name of the created J2C connection factory.`
	`description - The description for the created J2C connection factory.`
	`authDataAlias - the authentication data alias of the created J2C connection factory.`
	
	`Steps:`
	`None`
	
	The target object is a configuration object name of a J2C resource adapter.

options	is the option string for the invoked administrative command. Depending on which administrative command you are invoking, the administrative command can have required or optional option values. The options string is different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as options in the option string. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. If the invoked administrative command includes target objects, arguments, or steps, then the –interactive option is available to enter interactive mode. For example, using the output of the following online help for listDataSource:
	`WASX8006I: Detailed help for command: createJ2CConnectionFactory`
	
	`Description: Create a J2C connection factory`
	
	`*Target object: The parent J2C resource adapter of the created J2C connection factory.`
	
	`Arguments:`
	`*connectionFactoryInterface - A connection factory interface that is defined in the deployment description of the parent J2C resource adapter.`
	`*name - The name of the J2C connection factory.`
	`*jndiName - The JNDI name of the created J2C connection factory.`
	`description - The description for the created J2C connection factory.`
	`authDataAlias - the authentication data alias of the created J2C connection factory.`
	
	`Steps:`
	`None`
	
	Option names are specified with a dash before the names. The required options for this administrative command include: `-connectionFactoryInterface`, `-name`, and `-jndiName`. The optional options include: `-description` and `-authDataAlias`. In addition, you can also use the `-interactive` option. Options are specified in the option string, which is enclosed by a pair of braces ({}) in Jacl and a pair of brackets ([]) in Jython.

Example

- The following example invokes an administrative command with no target object, argument, or step:

  Using Jacl:

  ```
  $AdminTask listNodes
  ```

  Using Jython:

  ```
  print AdminTask.listNodes()
  ```

  Example output:

  ```
  myNode
  ```

- The following example invokes an administrative command with a target object string:

  Using Jacl:

```
set s1 [$AdminConfig getid /Server:server1/]
$AdminTask showServerInfo $s1
```

Using Jython:

```
s1 = AdminConfig.getid('/Server:server1/')
print AdminTask.showServerInfo(s1)
```

Example output:

```
{cell myCell}
{serverType APPLICATION_SERVER}
{com.ibm.websphere.baseProductVersion 6.0.0.0}
{node myNode}
{server server1}
```

- The following example invokes an administrative command with an option string:

  Using Jacl:

  ```
  $AdminTask getNodeMajorVersion {-nodeName myNode}
  ```

  Using Jython:

  ```
  print AdminTask.getNodeMajorVersion('[-nodeName myNode]')
  ```

  Example output:

  ```
  6
  ```

- The following example invokes an administrative command with a target object and non-step option strings:

  Using Jacl:

  ```
  set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
  $AdminTask createJ2CConnectionFactory $ra {-name myJ2CCF -jndiName j2c/cf -connectionFactoryInterface
  javax.resource.cci.ConnectionFactory}
  ```

  Using Jython:

  ```
  ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
  AdminTask.createJ2CConnectionFactory(ra, '[-name myJ2CCF -jndiName j2c/cf -connectionFactoryInterface
  javax.resource.cci.ConnectionFactory]')
  ```

  Example output:

  ```
  myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
  ```

- The following example invokes an administrative command with a target object and a step option:

  Using Jacl:

  ```
  set serverCluster [$AdminConfig getid /ServerCluster:myCluster/]
  $AdminTask createClusterMember $serverCluster {-memberConfig {{myNode myClusterMember "" "" false false}}}
  ```

  Using Jython:

  ```
  serverCluster = AdminConfig.getid('/ServerCluster:myCluster/')
  AdminTask.createClusterMember(serverCluster, '[-memberConfig [[myNode myClusterMember "" "" false false]]]')
  ```

  Example output:

  ```
  myClusterMember(cells/myCell/nodes/myNode|cluster.xml#ClusterMember_3673839301876)
  ```

Invoking an administrative command in interactive mode

These steps demonstrate how to invoke an administrative command in interactive mode.

About this task

To invoke an administrative command in batch mode, see “Invoking an administrative command in batch mode” on page 64.

1. Invoke the AdminTask object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Invoke an administrative command in interactive mode by issuing one of the following commands:
   - Use the following command invocation to enter interactive mode without providing another input in the command invocation:

Using Jacl:

```
$AdminTask commandName {-interactive}
```

Using Jython:

```
AdminTask.commandName('[-interactive]')
```

where:

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
-interactive	is the interactive option

- Use the following command invocation to enter interactive mode using an administrative command that takes a target object. You do not have to provide a target object to enter interactive mode. Target objects provided in the command invocation will be applied to the command and displayed as the current target object value during interactive prompting.

  Using Jacl:

  ```
  $AdminTask commandName targetObject {-interactive}
  ```

  Using Jython:

  ```
  AdminTask.commandName(targetObject, '[-interactive]')
  ```

  where:

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
targetObject	is the target object string for the invoked administrative command. The target object is different for each administrative command. View the online help for the invoked administrative command to learn more about what to specify as a target object.
-interactive	is the interactive option

- Use the following command invocation to enter interactive mode for an administrative command that takes options. You do not have to provide other options to enter interactive mode. Options provided in the command invocation are applied to the command and the option values will be displayed as the current values during interactive prompting.

  Using Jacl:

  ```
  $AdminTask commandName {-interactive commandOptions}
  ```

  Using Jython:

  ```
  AdminTask.commandName('[-interactive commandOptions]')
  ```

  where:

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
-interactive	is the interactive option

commandOptions	is the command option that is available for the associated administrative command. Available command options are different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as command options. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. For example, using the output of the following online help for the createJ2CConnectionFactory command:
	`WASX8006I: Detailed help for command: createJ2CConnectionFactory`
	
	`Description: Create a J2C connection factory`
	
	`*Target object: The parent J2C resource adapter of the created J2C connection factory.`
	
	`Arguments:`
	`*connectionFactoryInterface - A connection factory interface that is defined in the deployment description of the parent J2C resource adapter.`
	`*name - The name of the J2C connection factory.`
	`*jndiName - The JNDI name of the created J2C connection factory.`
	`description - The description for the created J2C connection factory.`
	`authDataAlias - the authentication data alias of the created J2C connection factory.`
	
	`Steps:`
	`None`
	
	In this example, five options are available:
	• -connectionFactoryInterface
	• -name
	• -jndiName
	• -description
	• -authDataAlias
	Each option requires a value. Three of the options are required and are denoted with a star (*).

- Use the following command invocation to enter interactive mode for an administrative command that has a target object and options. You do not have to specify a target object to enter interactive mode. The values specified are applied to the command before the command data is displayed. As a result, the values specified will be displayed as the current values during interactive prompting.

  Using Jacl:

  `$AdminTask commandName targetObject {-interactive commandOptions}`

  Using Jython:

`AdminTask.commandName(targetObject, '[-interactive commandOptions]')`

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
targetObject	is the target object string for the invoked administrative command. The expect target object varies with each admin command. Consult the online help on the invoked administrative command to learn more about what to specify as target object.
`-interactive`	is the interactive option

commandOptions	is the command option that is available for the associated administrative command. Available command options are different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as command options. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. For example, using the output of the following online help for the createJ2CConnectionFactory command:
	`WASX8006I: Detailed help for command: createJ2CConnectionFactory`
	
	`Description: Create a J2C connection factory`
	
	`*Target object: The parent J2C resource adapter of the created J2C connection factory.`
	
	`Arguments:`
	`*connectionFactoryInterface - A connection factory interface that is defined in the deployment description of the parent J2C resource adapter.`
	`*name - The name of the J2C connection factory.`
	`*jndiName - The JNDI name of the created J2C connection factory.`
	`description - The description for the created J2C connection factory.`
	`authDataAlias - the authentication data alias of the created J2C connection factory.`
	
	`Steps:`
	`None`
	
	In this example, five options are available:
	• -connectionFactoryInterface
	• -name
	• -jndiName
	• -description
	• -authDataAlias
	
	Each option requires a value. Three of the options are required and are denoted with a star (*).

Example

- The following example invokes an administrative command in interactive mode by specifying the -interactive option:

  Using Jacl:

  ```
  $AdminTask createJ2CConnectionFactory {-interactive}
  ```

  Using Jython:

  ```
  AdminTask.createJ2CConnectionFactory('[-interactive]')
  ```

  Example output:

  ```
  Create a J2C connection factory

  *The J2C resource adapter: "WebSphere Relational ResourceAdapter
  (cells/myCell/nodes/myNode|resources.xml#builtin_rra)"

  A connection factory
  interface (connectionFactoryInterface):javax.resource.cci.ConnectionFactory
  *Name (name): myJ2CCF
  *The JNDI name (jndiName): j2c/cf
  Description (description):
  authentication data alias (authDataAlias):

  create J2C connection factory

  F (Finish)
  C (Cancel)

  Select [F, C]: [F]

  myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
  ```

- The following example invokes an administrative command using the –interactive option with a target object that is specified in the command invocation:

  Using Jacl:

  ```
  set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
  $AdminTask createJ2CConnectionFactory $ra {-interactive}
  ```

  Using Jython:

  ```
  ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
  AdminTask.createJ2CConnectionFactory(ra, '[-interactive]')
  ```

  Example output:

  ```
  Create a J2C connection factory

  *The J2C resource adapter: ["WebSphere Relational ResourceAdapter
  (cells/myCell/nodes/myNode|resources.xml#builtin_rra)"]

  A connection factory interface (connectionFactoryInterface):
  javax.resource.cci.ConnectionFactory
  *Name (name): myJ2CCF
  *The JNDI name (jndiName): j2c/cf
  Description (description):
  authentication data alias (authDataAlias):

  create J2C Connection Factory

  F (Finish)
  C (Cancel)

  Select [F, C]: [F]

  myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
  ```

- The following example invokes an administrative command using the –interactive option where both the target object and the additional command options are specified in the command invocation:

Using Jacl:

```
set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
$AdminTask createJ2CConnectionFactory $ra {-name myNewCF -interactive}
```

Using Jython:

```
ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
AdminTask.createJ2CConnectionFactory(ra, '[-name myNewCF -interactive]')
```

Example output:

```
Create a J2C connection factory

*The J2C resource adapter: ["WebSphere Relational ResourceAdapter
(cells/myCell/nodes/myNode|resources.xml#builtin_rra)"]

A connection factory interface (connectionFactoryInterface):javax.resource.cci.ConnectionFactory
*Name (name): [myNewCF]
*The JNDI name (jndiName): j2c/cf
Description (description):
authentication data alias (authDataAlias):

create J2C Connection Factory

F (Finish)
C (Cancel)

Select [F, C]: [F]

myNewCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_3839439380269)
```

Administrative command interactive mode environment:

An administrative command can be run in interactive mode by providing the -interactive option in the options string when invoking the command.

You can still provide other options, even when using the interactive option. The options values that are specified are applied to the command before the command data is displayed. Whether or not other options are specified, the wsadmin tool steps the user through the command to collect command information.

The general interactive flow sequence is:

1. Collect user inputs for target object and parameters
2. If the command does not include a step, the command execution menu displays to run or cancel the command.
3. If the command includes a step, the menu to select the step displays. When all the required inputs are entered, the menu includes command execution.
4. When a step is selected, if the step supports collection, then the menu to select an object in the collection displays and you can exit the step. If you exit the step, repeat steps 1-3.
5. Collect user inputs for the selected step or for an object in the collection
6. Repeat steps 4 and 5 if from the collection step menu
7. Repeat steps 3-5 if from step selection menu

Depending on what input area is enabled by an administrative command, you can go through part or all of the interactive flow sequence. If an administrative command is run in interactive mode, the syntax to run the command except for the deletion of collection object in batch mode is generated and logged as a WASX7278I message in both the interactive session and in the wsadmin trace file.

Collect user inputs for target object and parameters

The following interactive prompt is used to collect inputs for the Target object and Arguments input areas that are displayed in the command-specific help:

```
Command title

Command Description

*target object title [current or default value]:
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

This screen is usually the first interactive screen that is displayed when an administrative command is invoked interactively unless the invoked command does not contain any target object and non-step command parameters. If a command does not have a target object, then the prompt for the target object is skipped. The number of parameters depends on the number of arguments in the Argument area of the command-specific help. If an input is required, then an asterisk (*) is placed in front of the title. The parameter name is displayed for information and is the name that is used to set this parameter in batch mode. If a parameter value is restricted to a set of values, then the valid choices are displayed. If current or default value is available, it is displayed. You can accept the existing value by pressing the Enter key. To add or change an existing value, enter a new value and click Enter.

Display command execution menu

If an administrative command does not contain a step, you are presented with the following menu after collecting values for target object and parameters:

```
Command title

F (Finish)
C (Cancel)

Select [F, C]: F
```

The Finish option runs the command and the Cancel option cancels the command. The default selection is F (Finish). This menu is the last menu that is displayed for a non-step command to exit interactive mode by either canceling or running the command.

Display command step selection and execution menu

If an administrative command contains a step, the following menu is displayed after collecting values for target object and parameters:

```
Command title
Command description
 -> *1. step1 title (step1 name)
     2. step2 title (step2 name)
    *3. step3 title (step3 name)
    (4. step4 title (step4 name))
     ...
     n. stepn title (stepn name)

S (Select)
N (Next)
P (Previous)
F (Finish)
C (Cancel)
H (Help)

Select [S, N, P, F, C, H]: S
```

The number of steps that is displayed in the menu depends on the administrative command. The step name is displayed for information and is the name that is used to set data in this step in batch mode. The following notations are used to describe a step:

- A "->" before the step indicates the current step position.

- A “*” before the step indicates a required step.
- A ( ) enclosing the entire step indicates a disabled step. You cannot navigate to this step by using the Next or Previous options.

Using the menu, you can navigate through steps sequentially by selecting Previous or Next. Select selects the current step, Finish runs the command, Cancel cancels the command, and Help provides online help for the command. Not all menu choices are available. Previous is not available if the current step is the first step. Next is not available if the current step is the last step. Finish is not available if still steps are still missing required inputs. The default selection is `S` (Select) if the current step is a valid step and steps are missing required inputs. Default selection is `F` (Finish) if all the required input is provided for the steps.

For commands with steps, you can exit interactive mode on this menu by either canceling or running the command.

Display collection step menu

A step might or might not support collection. A collection refers to objects of the same type. In an administrative command, a collection contains objects that have the same set of parameters. If a step that supports collection is selected, the wsadmin tool displays the following menu to add and select an object in the collection:

```
Step title (step name)
    | key param1 title (key param1 name), key param2 title (key param2 name), ...
    ------------------------------------------------------------------------
->  | object1 key param1 value, key param2 value, ...
   *| object2 key param1 value, key param2 value, ...
    ...
key param1 title, key param2 title, ... must be provided to specify a row in batch row.

S (Select Row)
N (Next)
P (Previous)
A (Add Row or Add Row Before)
D (Delete Row)
F (Finish)
H (Help)

Select [S, N, P, A, D, F, H]: F
```

The number of objects that display in the menu depends on the command step. Key parameters are identified by the step to use to uniquely identify an object in a collection. Key parameter values are displayed to identify an object to select. As with the command step selection menu, an arrow (->) is used to indicate the current object position, and a asterisk (*) is used to indicate that required input is missing in the object.

Use the menu to navigate through objects sequentially by selecting Previous or Next. Select Row selects the current object, Add Row adds a new object, Add Row Before adds a new object before the current object, Delete Row deletes the current object, Finish returns control back to the step selection and execution menu, and Help provides on-line help for the step. Not all menu choices are available. Previous is not available if there is no object in the collection or the first object is the current object. Next is not available if there is no object in the collection or the last object is the current object. Select Row is available only if there is a current object. Add Row is provided only if there is no object in the collection and the step supports new object to be added. Add Row Before is provided if the step supports new object to be added and there are existing objects in the collection. Delete Row is provided only if there is a current object and the step supports an object to be deleted. Finish is not available if there are still objects missing required inputs. Default selection is A (Add Row) when there is no object in the collection and the step supports objects to be added. Default selection is S (Select Row) if there is a current object and there are still objects missing required inputs. Default selection is F (Finish) if there is no required input missing in any object.

Collect user inputs for parameters of a collection object

After a collection object is selected, the parameter value for each parameter is prompted sequentially as shown in the following example:

```
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

The number of parameters depends on the number of arguments in the Argument area of the command step-specific help. The same asterisk (*) notation is used to denote a required parameter. If a parameter value is restricted to a set of values, then the valid choices are displayed. If the current or default value is available, it is displayed. For each writable parameter, you can accept the existing value by pressing Enter. To add or change an existing value, enter a new value and press Enter. For a read-only parameter, the parameter and its value are displayed. You will not be given the prompt to modify its value. After you go through all of the parameters, the wsadmin tool returns to the collection step menu.

Collect user inputs for non-collection step

This step has two parts. The first part displays the current or default parameter values for the selected step, as shown in the following example:

```
Step title (step name)

*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...

Select [C (Cancel), E (Edit)]: [E]
```

No prompting is included in this part. Instead, this part is more like a help function providing parameter information on the selected step. The number of parameters depends on the number of arguments in the `argument` area of the command step specific help. The asterisk (*) notation denotes a required parameter. If a parameter value is restricted to a set of values, then the valid choices will be displayed. If the current or default value is available, it is displayed. You can choose to cancel the step or continue to the next part to provide parameter inputs. The default selection is `Edit`. Because it is possible that you are seeing default values assigned to a new piece of data that is not yet set in the step, you can accept the default selection to continue to the next part. Otherwise, if no data exists in the selected step, selecting Cancel does not result in creating the data.

If you accept the default `Edit` selection, collect user inputs for parameters sequentially just like Collect user inputs for parameters of a collection object.

```
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

For each writable parameter, you can accept the existing value by pressing Enter. To add or change an existing value, enter a new value and then press Enter. For a read-only parameter, the parameter and its value are displayed. You will not be given the prompt to modify the value of the parameter. As soon as you step through all the parameters, the wsadmin tool will lead you back to the command step selection and execution menu.

Data types for the AdminTask object

The parameters for the AdminTask object accept various data types for different commands. This topic provides examples of valid data type syntax.

The following table lists the primitive and Java data types that the AdminTask object accepts for different commands, including the following data types:

- String

- Boolean
- Character
- Integer
- Long
- Byte
- Short
- Float
- Double
- Javax.management.ObjectName
- Java.util.Properties
- String[]
- Integer[]
- Jaca.net.URL
- Javax.management.Attribute
- Javax.management.AttributeList
- Java.util.ArrayList
- Java.util.List
- Java.util.Hashtable

The following example command specifies various data types for parameter values that are commonly used with the AdminTask object:

```
wsadmin>AdminTask.helloWorld('[-personName John -personalInfo [ [cellPhone 123-456-7890] [workPhone 123-456-7892]
 [homePhone 123-456-7891] ] -pets [dog cat] -personID WebSphere:John(organization=ibm,country=usa,state=texas,city=austin)
 –personAttrs [ [gender male] [age 29] [citizership USA] ] -hobbyList [swim tennis baseball]
 -favorFoodTable [ [juice orange] [fruit apple] ] ]')
```

where:

Parameter	Data type	Example value
personName	String	John
personalInfo	java.util.Properties	[[cellPhone 123-456-7890] [workPhone 123-456-7892] [homePhone 123-456-7891]]
pets	String[]	[dog cat]
personID	javax.management.ObjectName	WebSphere:John(organization=ibm,country=usa,state=texas,city=austin)
personAttrs	javax.management.AttributeList	[[gender male] [age 29] [citizership USA]]
hobbyList	java.util.ArrayList	[swim tennis baseball]
favorFoodTable	java.util.Hashtable	[[juice orange] [fruit apple]]

Starting the wsadmin scripting client

You can use the wsadmin tool to configure and administer application servers, application deployment, and server run-time operations.

About this task

The wsadmin tool provides the ability to automate configuration tasks for your environment by running scripts. However, there are some limitations for using the wsadmin tool, including:

- The wsadmin tool only supports the Jython and Jacl scripting languages. The Version 6.1 release of WebSphere Application Server represented the start of the deprecation process for the Jacl syntax that is associated with the wsadmin tool. The Jacl syntax for the wsadmin tool continues to remain in the product and is supported for at least two major product releases. After that time, the Jacl language support might be removed from the wsadmin tool. The Jython syntax for the wsadmin tool is the

strategic direction for WebSphere Application Server administrative automation. The application server provides significantly enhanced administrative functions and tooling that support product automation and the use of the Jython syntax.

- The wsadmin tool manages the installation, configuration, deployment, and runtime operations for application servers, deployment managers, administrative agents, and job managers that run the same version or a higher version of the product. The wsadmin tool cannot connect to an application server, deployment manager, administrative agent, or job manager that runs a product version which is older than the version of the wsadmin tool. For example, a Version 6.x wsadmin client cannot connect to a Version 5.x application server. However, a Version 5.x wsadmin client can connect to a Version 6.x application server. This limitation exists because new functionality is added to the wsadmin tool in each product release. You cannot use new command functionality on application servers running previous product versions.
- The wsadmin tool operates at the deployment manager node level in a mixed-cell environment. Do not run wsadmin at the application server node level to ensure that all command functionality is available.

In a flexible management environment, you can connect the wsadmin tool to a base application server, deployment manager, administrative agent, or job manager process. If you do not specify the port of the base application server or the profile name assigned to the job manager, the wsadmin tool automatically connects to the administrative agent.

1. Locate the command that starts the wsadmin scripting client.

   Choose one of the following:

   - Invoke the scripting process using a specific profile. The QShell command for invoking a scripting process is located in the *profile_root*/bin directory. The name of the QShell script is `wsadmin`. If you use this option, you do not need to specify the `-profileName` *`profilename`* parameter.
   - Invoke the scripting process using the default profile. The wsadmin Qshell command is located in the *app_server_root*/bin directory. If you do not want to connect to the default profile, you must specify the `-profileName` *`profilename`* parameter to indicate the profile that you want to use.

2. In a flexible management environment, determine whether to connect to a base application server, administrative agent, or job manager process.

   - Connect to the administrative agent process.

     Connect the wsadmin tool to the administrative agent to configure, manage, and administer servers. If you do not specify connection options, the wsadmin tool automatically connects to the administrative agent process. Use the following command to connect to the administrative agent:

     ```
     wsadmin -lang jython
     ```

   - Connect to a base application server process.

     Connect the wsadmin tool to a base application server to manage settings for a specific server of interest. Use this connection type when connecting to a node that contains one server and is registered with the administrative agent. Use the following command to connect to a base application server:

     ```
     wsadmin -conntype SOAP [-port 4213] -lang jython
     ```

   - Connect to the job manager process.

     Connect the wsadmin tool to the job manager to submit, monitor, and manage administrative jobs. Use the following command to connect to the job manager:

     ```
     wsadmin -profileName myJobManager -lang jython
     ```

3. Review additional connection options for the wsadmin tool.

   You can start the wsadmin scripting client in several different ways. To specify the method for running scripts, perform one of the following wsadmin tool options:

   Run scripting commands interactively

   Run wsadmin with an option other than `-f` or `-c` or without an option. The wsadmin tool starts and displays an interactive shell with a wsadmin prompt. From the wsadmin prompt, enter any Jacl or Jython command. You can also invoke commands using the AdminControl, AdminApp,

AdminConfig, AdminTask, or Help wsadmin objects.To leave an interactive scripting session, use the **quit** or **exit** commands. These commands do not take any arguments.

The following examples launch the wsadmin tool:

- Launch the wsadmin tool using Jython:

```
wsadmin -lang jython
```

- Launch the wsadmin tool using Jython when security is enabled:

```
wsadmin -lang jython -user wsadmin

 -password wsadmin
```

- Launch the wsadmin tool using Jacl with no options:

```
wsadmin -lang jacl
```

Run scripting commands as individual commands

Run the wsadmin tool with the -c option.

The following examples run commands individually:

- Run the list command for the AdminApp object using Jython:

```
wsadmin -lang jython -c "AdminApp.list()"
```

- Run the list command for the AdminApp object using Jacl:

```
wsadmin -c "$AdminApp list"
```

Run scripting commands in a script

Run the wsadmin tool with the `-f` option, and place the commands that you want to run into the file.

The following examples run scripts:

- Run the a1.py script using Jython:

```
wsadmin -lang jython -f  al.py
```

where the `al.py` file contains the following commands:

```
apps = AdminApp.list()
print apps
```

Run scripting commands in a profile script

A *profile script* is a script that runs before the main script, or before entering interactive mode. You can use profile scripts to set up a scripting environment that is customized for the user or the installation.

By default, the following profile script files might be configured for the `com.ibm.ws.scripting.profiles` profiles property in the *app_server_root*`/properties/wsadmin.properties` file:

app_server_root`/bin/securityProcs.jacl`
app_server_root`/bin/LTPA_LDAPSecurityProcs.jacl`

By default, these files are in ASCII. If you use the `profile.encoding` option to run EBCDIC encoded profile script files, change the encoding of the files to EBCDIC.

To run scripting commands in a profile script, run the wsadmin tool with the -profile option, and include the commands that you want to run into the profile script.

To customize the script environment, specify one or more profile scripts to run.

Do not use parenthesis in node names when creating profiles.

The following examples run profile scripts:

- Run the a1prof.py script using Jython:

```
wsadmin -lang jython -profile alprof.py
```

where the `alprof.py` file contains the following commands:

```
apps = AdminApp.list()
print "Applications currently installed:\n " + apps
```

- Run the a1prof.py script using Jacl:

```
wsadmin -profile alprof.jacl
```

where the `alprof.jacl` file contains the following commands:

```
set apps [$AdminApp list]
puts "Applications currently installed:\n$apps"
```

Results

The wsadmin returns the following output when it establishes a connection to the server processs:

Jthon example output:

```
WASX7209I: Connected to process server1
on node myhost using SOAP connector;
The type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

Jacl example output:

```
WASX7209I: Connected to process server1
on node myhost using SOAP connector;
The type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

Restricting remote access using scripting

You can use the wsadmin tool to restrict remote administration so that administrators only manage nodes locally. This prevents the base node from opening remote ports for the administrator. Each administrative connection must occur from the local workstation.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Restrict remote access.

   Run the following set of commands for each server of interest to restrict remote access:

```
server=AdminConfig.getId('/Server:server1/')
AdminTask.setAdminProtocolEnabled(server, '[-conntype SOAP –enable false]')
AdminTask.setAdminProtocolEnabled(server, '[-conntype RMI –enable false]')
AdminTask.setAdminProtocolEnabled(server, '[-conntype JSR160RMI –enabled false]')
AdminTask.setAdminProtocol(server,'[-conntype IPC -mode local]')
```

3. Restart each server.

   Use the stopAllServers and startAllServers commands in the AdminServerManagement script library to restart each server configured with local access only, as the following example demonstrates:

```
AdminServerManagement.stopAllServers("myNode")
AdminServerManagement.startAllServers("myNode")
```

Chapter 3. Using the script library to automate the application serving environment

The script library provides Jython script procedures to assist in automating your environment. Use the sample scripts to manage applications, resources, servers, nodes, and clusters. You can also use the script procedures as examples to learn the Jython syntax.

About this task

Note: The Jython script library provides a set of procedures to automate the most common application server administration functions. For example, you can use the script library to easily configure servers, applications, mail settings, resources, nodes, business-level applications, clusters, authorization groups, and more. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Each script from the script library directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory, and save existing automation scripts in the *app_server_root*/scriptLibraries directory. Each script library name must be unique and cannot be duplicated.

Note: Do not edit the script procedures in the script library. To customize script library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library.

To automatically load Jython scripts (*.py) that are not located in the *app_server_root*/scriptLibraries directory when the wsadmin tool starts, set the `wsadmin.script.libraries` system property to the script location. For example, if your script libraries are saved in the temp directory on a Windows operating system, the following example sets the script path in the wsadmin command line tool:

```
bin>wsadmin -lang jython -javaoption "-Dwsadmin.script.libraries=c:/myJythonScripts"
```

To load multiple directories, specify each directory in the system property separated by a semicolon (;), as the following example demonstrates:

```
bin>wsadmin -lang jython -javaoption "-Dwsadmin.script.libraries=c:/myJythonScripts;c:/AdminScripts;c:/configScripts"
```

The script library provides automation scripts for the following application server administration functions:

- Manage application servers. You can use the AdminServerManagement scripts to configure classloaders, Java virtual machine (JVM) settings, Enterprise JavaBean (EJB) containers, performance monitoring, dynamic cache, and so on.
- Manage server and system architecture. You can use the AdminServerManagement, AdminNodeManagement, and AdminClusterManagement script libraries to manage clusters, nodes, and node groups.
- Manage applications. You can use the AdminApplication scripts to install, uninstall, and update your applications with various options.
- Manage data access resources. You can use the AdminJDBC and AdminJ2C script libraries to manage data sources and Java Database Connectivity (JDBC) providers, and to create and configure Java 2 Connector (J2C) resource adapters.
- Manage messaging resources. You can use the AdminJMS script library to configure and manage your Java Messaging Service (JMS) configurations.
- Manage mail resources. You can use the AdminResources scripts in the script library to configure mail, URL, and resource settings.
- Managing authorization groups. You can use the AdminAuthorizations scripts to configure authorization groups.
- Monitor performance and troubleshoot configurations. You can use the AdminUtilities scripts to configure trace, debugging, logs, and performance monitoring.
- Get script library help using wsadmin You can use the AdminLibHelp script library to list each available script library, display information for specific script libraries, and to display information for specific script procedures.

What to do next

Determine which scripts to use to automate your environment, or create custom scripts using assembly tools.

Automating server administration using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the server management scripts to configure servers, the server runtime environment, Web containers, performance monitoring, and logs. You can also use the scripts to administer your servers.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

```
#
# My Custom Jython Script - file.py
#
AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
    "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The AdminServerManagement procedures in scripting library are located in the *app_server_root*/scriptLibraries/servers/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminServerManagement.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. Use the following steps to create an application server, connect the application server to the AdminService interface, configure Java virtual machine (JVM) settings, add the application server to a cluster, and propagate the changes to the node.

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create an application server.

   Run the createApplicationServer script procedure from the AdminServerManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminServerManagement.createApplicationServer("myNode", "myServer", "default")"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
```

3. Connect the application server of interest to the AdminService interface.

   The AdminService interface is the server interface to the application server administration functions. To connect the application server to the AdminService interface, run the configureAdminService script procedure from the AdminServerManagement script library, specifying the node name, server name, and connector type arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminServerManagement.configureAdminService("myNode", "myServer", "IPC", "JSR160RMI")
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.configureAdminService("myNode", "myServer", "IPC", "JSR160RMI")
```

4. Configure the Java virtual machine (JVM).

   As part of configuring an application server, you might define settings that enhance the way your operating system uses of the JVM. The JVM is an interpretive computing engine responsible for running the byte codes in a compiled Java program. The JVM translates the Java byte codes into the native instructions of the host machine. The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it.

   Run the configureJavaVirtualMachine script procedure from the AdminServerManagement script library, specifying the node name, server name, whether to run the JVM in debug mode, and any debug arguments to pass to the JVM process. You can optionally specify additional configuration attributes with an attribute list. Use the following example to configure the JVM:

```
bin>wsadmin -lang jython -c "AdminServerManagement.configureJavaVirtualMachine("myNode", "myServer", "true", "mydebug", [["internalClassAccessMode", "RESTRICT"],
    ["disableJIT", "false"], ["verboseModeJNI", "false"]])"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.configureJavaVirtualMachine("myNode", "myServer", "true", "mydebug", [["internalClassAccessMode", "RESTRICT"],
   ["disableJIT", "false"], ["verboseModeJNI", "false"]])
```

5. Create a cluster, and add the application server as a cluster member.

   Run the createClusterWithFirstMember script procedure from the AdminClusterManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "myServer")"
wsadmin>AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "myServer")
```

6. Synchronize the node.

   To propagate the configuration changes to the node, run the syncNode script procedure from the AdminNodeManagement script library, and specify the node of interest, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminNodeManagement.syncNode("myNode")"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminNodeManagement.syncNode("myNode")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Server settings configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to configure class loaders, Java Virtual Machine (JVM) settings, Enterprise JavaBean (EJB) containers, performance monitoring, dynamic cache, and so on. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V70 directory.

Use the following script procedures to administer your application server:

- “configureAdminService” on page 86
- “configureApplicationServerClassloader” on page 86
- “configureDynamicCache” on page 87
- “configureEJBContainer” on page 87
- “configureFileTransferService” on page 88
- “configureListenerPortForMessageListenerService” on page 88
- “configureMessageListenerService” on page 89
- “configureStateManageable” on page 89

Use the following script procedures to configure your application server runtime environment:

- “configureCustomProperty” on page 90
- “configureCustomService” on page 90
- “configureEndPointsHost” on page 91
- “configureJavaVirtualMachine” on page 91
- “configureORBService” on page 91
- “configureProcessDefinition” on page 92
- “configureRuntimeTransactionService” on page 92
- “configureThreadPool” on page 93
- “configureTransactionService” on page 94
- “setJVMProperties” on page 94
- “setTraceSpecification” on page 95

Use the following script procedures to configure Web containers for your application server:

- “configureCookieForServer” on page 95
- “configureHTTPTransportForWebContainer” on page 96
- “configureSessionManagerForServer” on page 96
- “configureWebContainer” on page 97

Use the following script procedures to configure logs and monitor performance for your application server:

- “configureJavaProcessLogs” on page 98
- “configurePerformanceMonitoringService” on page 98
- “configurePMIRequestMetrics” on page 99
- “configureServerLogs” on page 100

• “configureTraceService” on page 100

configureAdminService

This script configures settings for the AdminService interface. The AdminService interface is the server-side interface to the application server administration functions.

To run the script, specify the node name, server name, local connection protocol, and remote connection protocol, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
localAdminProtocol	Specifies the type of connector to use to connect the AdminService interface to the application server for local connection.
remoteAdminProtocol	Specifies the type of connector to use to connect the AdminService interface to the application server for remote connection.
otherAttributeList	Optionally specifies additional attributes in the following format: `[[″enabled″, ″true″], [″pluginConfigService″, ″(cells/timmieNode02Cell/nodes/timmieNode01/servers/server1|server.xml#PluginConfigService_1183122130078)″]]`

Syntax

```
AdminServerManagement.configureAdminService(nodeName, serverName, localAdminProtocol, remoteAdminProtocol, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureAdminService("myNode", "myServer", "IPC", "SOAP",
    [["enabled", "true"], ["pluginConfigService",
       "(cells/timmieNode02Cell/nodes/timmieNode01/servers/server1|server.xml#PluginConfigService_1183122130078)"]])
```

configureApplicationServerClassloader

This script configures a class loader for the application server. Class loaders enable applications that are deployed on the application server to access repositories of available classes and resources.

To run the script, specify the node name, server name, policy, mode, and library name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
policy	Specifies the application class loader policy as `SINGLE` or `MULTIPLE`. Specify the `SINGLE` value to prevent the isolation applications, and to configure the application server to use a single application class loader to load all of the EJB modules, shared libraries, and dependency Java archive (JAR) files in the system. Specify the `MULTIPLE` value to isolate applications and provide each application with its own class loader to load EJB modules, shared libraries, and dependency JAR files.
mode	Specifies the class loader mode as `PARENT_FIRST` or `APPLICATION_FIRST`. The `PARENT_FIRST` option causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. The `APPLICATION_FIRST` option causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.
libraryName	Specifies the name of the shared library of interest.

Syntax

```
AdminServerManagement.configureApplicationServerClassloader(nodeName, serverName, policy, mode, libraryName)
```

Example usage

```
AdminServerManagement.configureApplicationServerClassloader("myNode", "MULTIPLE", "PARENT_FIRST", "myLibraryReference")
```

configureDynamicCache

This script configures the dynamic cache service in your server configuration. The dynamic cache service works within an application server JVM, intercepting calls to cacheable objects. For example, the dynamic cache service intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.

To run the script, specify the node name, server name, default priority, cache size, external cache group name, and external cache group type arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
defaultPriority	Specifies the default priority for cache entries, determining how long an entry stays in a full cache. Specify an integer between 1 and 255.
cacheSize	Specifies a positive integer as the value for the maximum number of entries that the cache holds. Enter a cache size value in this field that is between the range of 100 through 200000.
externalCacheGroupName	The external cache group name needs to match the ExternalCache property as defined in the servlet or JavaServer Pages (JSP) file cachespec.xml file. When external caching is enabled, the cache matches pages with its Universal Resource Identifiers (URI) and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of from the application server.
externalCacheGroupType	Specifies the external cache group type.
otherAttributeList	Optionally specifies additional configuration options for the dynamic cache service in the following format: `[[″cacheProvider″, ″myProvider″], [″diskCacheCleanupFrequency″, 2], [″flushToDiskOnStop″, ″true″]]`

Syntax

```
AdminServerManagement.configureDynamicCache(nodeName, serverName, defaultPriority,
    cacheSize, externalCacheGroupName, externalCacheGroupType,
    otherAttributeList)
```

Example usage

```
AdminServerManagement.configureDynamicCache("myNode", "myServer", 2, 5000, "EsiInvalidator",
   "SHARED", [["cacheProvider", "myProvider"], ["diskCacheCleanupFrequency", 2], ["flushToDiskOnStop","true"]])
```

configureEJBContainer

This script configures an Enterprise JavaBeans™ (EJB) container in your server configuration. An EJB container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean's operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

To run the script, specify the node name, server name, passivation directory, and default datasource Java Naming and Directory Interface (JNDI) name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
passivationDirectory	Specifies the directory into which the container saves the persistent state of passivated stateful session beans. This directory must already exist. It is not automatically created.
defaultDatasourceJNDIName	Specifies the JNDI name of a data source to use if no data source is specified during application deployment. This setting is not applicable for EJB 2.x-compliant container-managed persistence beans.

Syntax

```
AdminServerManagement.configureEJBContainer(nodeName, serverName, passivationDir, defaultDatasourceJNDIName)
```

Example usage

```
AdminServerManagement.configureEJBContainer("myNode", "myServer", "/temp/myDir", "jndi1")
```

configureFileTransferService

This script configures the file transfer service for the application server. The file transfer service transfers files from the deployment manager to individual remote nodes.

To run the script, specify the node name, server name, number of times to retry the file transfer, and the time to wait before retrying the file transfer, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
retriesCount	Specifies the number of times you want the file transfer service to retry sending or receiving a file after a communication failure occurs. The default value is `3`.
retryWaitTime	Specifies the number of seconds that the file transfer service waits before it retries a failed file transfer. The default value is `10`.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["enable", "true"]]`

Syntax

```
AdminServerManagement.configureFileTransferService(nodeName, serverName, retriesCount, retryWaitTime, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureFileTransferService("myNode", "myServer", 5, 600, [["enable", "true"]])
```

configureListenerPortForMessageListenerService

This script configures the listener port for the message listener service in your server configuration. The message listener service is an extension to the Java Messaging Service (JMS) functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

To run the script, specify the node name, server name, listener port name, connection factory JNDI name, destination JNDI name, maximum number of messages, maximum number of retries, and the maximum session arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
listenerPortName	The name by which the listener port is known for administrative purposes.
connectionFactoryJNDIName	The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`.
destinationJNDIName	The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`
maxMessages	The maximum number of messages that the listener can process in one transaction. If the queue is empty, the listener processes each message when it arrives. Each message is processed within a separate transaction.
maxRetries	The maximum number of times that the listener tries to deliver a message before the listener is stopped, in the range 0 through 2147483647. The maximum number of times that the listener tries to deliver a message to a message-driven bean instance before the listener is stopped.
maxSession	Specifies the maximum number of concurrent sessions that a listener can have with the JMS server to process messages. Each session corresponds to a separate listener thread and therefore controls the number of concurrently processed messages. Adjust this parameter when the server does not fully use the available capacity of the machine and if you do not need to process messages in a specific message order.

Syntax

```
AdminServerManagement.configureListenerPortForMessageListener(nodeName, serverName, listenerPortName, connectionFactoryJNDIName,
     destinationJNDIName, maxMessages, maxRetries, maxSession)
```

Example usage

```
AdminServerManagement.configureListenerPortForMessageListener("myNode", "myServer", "myListenerPort", "connJNDI", "destJNDI", 5, 2, 3)
```

configureMessageListenerService

This script configures the message listener service in your server configuration. The message listener service is an extension to the Java Messaging Service (JMS) functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

To run the script, specify the node name, server name, maximum number of message listener retries, listener recovery interval, pooling threshold, and pooling timeout attributes, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
maxListenerRetry	Specifies the maximum number of times that a listener port managed by this service tries to recover from a failure before giving up and stopping. When stopped the associated listener port is changed to the stop state.
listenerRecoveryInterval	Specifies the time in seconds between retry attempts by a listener port to recover from a failure.
poolingThreshold	Specifies the maximum number of unused connections in the pool. The default value is `10`.
poolingTimeout	Specifies the number of milliseconds after which a connection in the pool is destroyed if it has not been used. An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes.
otherAttributeList	Optionally specifies additional message listener attributes in the following format: `[[″description″, ″test message listener″], [″isGrowable″, ″true″], [″maximumSize″, 100], [″minimumSize″, 5]]`

Syntax

```
AdminServerManagement.configureMessageListenerService(nodeName, serverName, maxListenerRetry, listenerRecoveryInterval,
     poolingThreshold, poolingTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureMessageListenerService("myNode", "myServer", 5, 120, 20, 600000, "myProp", "myValue",
      [["description", "test message listener"], ["isGrowable", "true"], ["maximumSize", 100], ["minimumSize", 5]])
```

configureStateManageable

This script configures the initial state of the application server. The initial state refers to the desired state of the component when the server process starts.

To run the script, specify the node name, server name, parent type, and initial state arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the type of component to modify.
initialState	Specifies the desired state of the component when the server process starts. Valid values are `START` and `STOP`.

Syntax

```
AdminServerManagement.configureStateManageable(nodeName, serverName, parentType, initialState)
```

Example usage

```
AdminServerManagement.configureStateManageable("myNode", "myServer", "Server", "START")
```

configureCustomProperty

This script configures custom properties in your application server configuration. You can use custom properties for configuring internal system properties which some components use, for example, to pass information to a Web container.

To run the script, specify the node name, server name, parent type, property name, and property value arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the type of component to configure.
propertyName	Specifies the custom property to configure.
propertyValue	Specifies the value of the custom property to configure.
otherAttributeList	Optionally specifies additional attributes in the following format: `[[″commTraceEnabled″, ″true″], [″enable″, ″true″]]`

Syntax

```
AdminServerManagement.configureCustomProperty(nodeName, serverName, parentType, propertyName, propertyValue, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCustomProperty("myNode", "myServer", "ThreadPool", "myProp1", "myPropValue",
    [["description", "my property test"], ["required", "false"]])
```

configureCustomService

This script configures a custom service in your application server configuration. Each custom services defines a class that is loaded and initialized whenever the server starts and shuts down. Each of these classes must implement the com.ibm.websphere.runtime.CustomService interface. After you create a custom service, use the administrative console to configure that custom service for your application servers.

To run the script, specify the node name, server name, and preferred connector type, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
classname	Specifies the class name of the service implementation. This class must implement the Custom Service interface.
displayname	Specifies the name of the service.
classpath	Specifies the class path used to locate the classes and JAR files for this service.
otherAttributeList	Optionally specifies additional attributes in the following format: `[[″description″, ″test custom service″], [″enable″, ″true″]]`

Syntax

```
AdminServerManagement.configureCustomService(nodeName, serverName, classname, displayname, classpath, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCustomService("myNode", "myServer", "myClass", "myName", "/temp/boo.jar",
     [["description", "test custom service"], ["enable", "true"]])
```

configureEndPointsHost

This script configures the host name of the server endpoints. To run the script, specify the node name, server name, and host name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
hostName	Specifies the name of the host of interest.

Syntax

```
AdminServerManagement.configureEndPointsHost(nodeName, serverName, hostName)
```

Example usage

```
AdminServerManagement.configureEndPointsHost("myNode", "AppServer01", "myHostname")
```

configureJavaVirtualMachine

This script configures a Java virtual machine (JVM). The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it.

To run the script, specify the configuration ID of the JVM of interest, whether to enable debug mode, and additional debug arguments, as defined in the following table:

Argument	Description
javaVirtualMachineConfigID	Specifies the configuration ID of the Java virtual machine you want to make changes.
debugMode	Specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. If you set the debugMode argument to `true`, then you must specify debug arguments.
debugArgs	Specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["internalClassAccessMode", "RESTRICT""], ["disableJIT", "false"], ["verboseModeJNI", "false"]]`

Syntax

```
AdminServerManagement.configureJavaVirtualMachine(javaVirtualMachineConfigID, debugMode, debugArgs, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureJavaVirtualMachine
   ("(cells/WAS00Network/nodes/ndnode1/servers/server1|server.xml#JavaVirtualMachine_1208188803955)", "true",
   "mydebug", [["internalClassAccessMode", "RESTRICT"], ["disableJIT", "false"], ["verboseModeJNI", "false"]])
```

configureORBService

This script configures an Object Request Broker (ORB) service in your server configuration. An Object Request Broker (ORB) manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.

To run the script, specify the node name, server name, request timeout, request retry count, request retry delay, maximum connection cache, minimum connection cache, and locate request timeout arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
requestTimeout	Specifies the number of seconds to wait before timing out on a request message.
requestRetriesCount	Specifies the number of times that the ORB attempts to send a request if a server fails. Retrying sometimes enables recovery from transient network failures. This field is ignored on the z/OS® platform.
requestRetriesDelay	Specifies the number of milliseconds between request retries. This field is ignored on the z/OS platform.
connectionCacheMax	Specifies the maximum number of entries that can occupy the ORB connection cache before the ORB starts to remove inactive connections from the cache. This field is ignored on the z/OS platform. It is possible that the number of active connections in the cache will temporarily exceed this threshold value. If necessary, the ORB will continue to add connections as long as resources are available.
connectionCacheMin	Specifies the minimum number of entries in the ORB connection cache. This field is ignored on the z/OS platform. The ORB will not remove inactive connections when the number of entries is below this value.
locateRequestTimeout	Specifies the number of seconds to wait before timing out on a LocateRequest message. This field is ignored on the z/OS platform.
otherAttributeList	Optionally specifies additional attributes in the following format: `[[″commTraceEnabled″, ″true″], [″enable″, ″true″]]`

Syntax

```
AdminServerManagement.configureORBService(nodeName, serverName, requestTimeout, requestRetriesCount, requestRetriesDelay,
   connectionCacheMax, connectionCacheMin, locateRequestTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureMessageListenerService("myNode", "myServer", 5, 120, 20, 600000, 20, 300, [["commTraceEnabled", "true"], ["enable", "true"]])
```

configureProcessDefinition

This script configures the server process definition. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
otherParamList	Specifies additional parameters for the process definition configuration in the following format: `[[″executableName″, ″value1″], [″executableArguments″, ″value2″], [″workingDirectory″, ″value3″]]`

Syntax

```
AdminServerManagement.configureProcessDefintion(nodeName, serverName, otherParamList)
```

Example usage

```
AdminServerManagement.configureProcessDefinition("myNode", "myServer",
   [["executableName", "value1"],["executableArguments"."value2"],["workingDirectory","value3"]])
```

configureRuntimeTransactionService

This script configures the transaction service for your server configuration. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.

To run the script, specify the node name, server name, total transaction lifetime timeout , and client inactivity timeout arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
totalTranLifetimeTimeout	Specifies the default maximum time, in seconds, allowed for a transaction that is started on this server before the transaction service initiates timeout completion. Any transaction that does not begin completion processing before this timeout occurs is rolled back.
clientInactivityTimeout	Specifies the maximum duration, in seconds, between transactional requests from a remote client. Any period of client inactivity that exceeds this timeout results in the transaction being rolled back in this application server. If you set this value to 0, there is no timeout limit.

Syntax

```
AdminServerManagement.configureRuntimeTransactionService(nodeName, serverName, totalTranLifetimeTimeout, clientInactivityTimeout)
```

Example usage

```
AdminServerManagement.configureRuntimeTransactionService("myNode", "myServer", 600, 600)
```

configureThreadPool

This script configures thread pools in your server configuration. A thread pool enables components of the server to reuse threads, which eliminates the need to create new threads at run time. Creating new threads expends time and resources.

To run the script, specify the node name, server name, parent type, thread pool name, maximum size, minimum size, and the amount of time before timeout occurs, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the maximum number of times that a listener port managed by this service tries to recover from a failure before stopping. When stopped the associated listener port is changed to the stop state.
threadPoolName	Specifies the name of the thread pool of interest.
maximumSize	Specifies the maximum number of threads to maintain in the default thread pool. If your Tivoli® Performance Viewer shows the Percent Maxed metric to remain consistently in the double digits, consider increasing the Maximum size. The Percent Maxed metric indicates the amount of time that the configured threads are used.
minimumSize	Specifies the minimum number of threads to allow in the pool. When an application server starts, no threads are initially assigned to the thread pool. Threads are added to the thread pool as the workload assigned to the application server requires them, until the number of threads in the pool equals the number specified in the Minimum size field. After this point in time, additional threads are added and removed as the workload changes. However the number of threads in the pool never decreases below the number specified in the Minimum size field, even if some of the threads are idle.
inactivityTimeout	Specifies the number of milliseconds of inactivity that should elapse before a thread is reclaimed. A value of `0` indicates not to wait and a negative value (less than 0) means to wait forever.
otherAttributeList	Specifies additional configuration attributes in the following format: `[["description", "testing thread pool"], ["isGrowable", "true"], ["name", "myThreadPool"]]`

Syntax

```
AdminServerManagement.configureThreadPool(nodeName, serverName, parentType, threadPoolName, maximumSize, minimumSize, inactivityTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureThreadPool
   ("myNode", "myServer", "myThreadPool", "ObjectRequestBroker", 20, 5, 6000,
   [["description", "testing thread pool"], ["isGrowable", "true"],  ["name","myThreadPool"]])
```

configureTransactionService

This script configures the transaction service for your application server. You can use transactions with your applications to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent.

To run the script, specify the node name, server name, total transaction lifetime timeout, client inactivity timeout, maximum transaction timeout, heuristic retry limit, heuristic retry wait, propogate or BMT transaction lifetime timeout, and asynchronous response timeout arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the server name of the of interest.
totalTranLifetimeTimeout	Specifies the default maximum time, in seconds, allowed for a transaction that is started on this server before the transaction service initiates timeout completion. Any transaction that does not begin completion processing before this timeout occurs is rolled back. This timeout is used only if the application component does not set its own transaction timeout. **Note:** Only the total transaction lifetime timeout and the maximum transaction timeout have grace periods. You can disable the grace periods using the DISABLE_TRANSACTION_TIMEOUT_GRACE_PERIOD custom property.
clientInactivityTimeout	Specifies the maximum duration, in seconds, between transactional requests from a remote client. Any period of client inactivity that exceeds this timeout results in the transaction being rolled back in this application server. If you set this value to 0, there is no timeout limit.
maximumTransactionTimeout	Specifies the upper limit of the transaction timeout, in seconds, for transactions that run in this server. This value should be greater than or equal to the total transaction timeout. This timeout constrains the upper limit of all other transaction timeouts.
heuristicRetryLimit	Specifies the number of times that the application server retries a completion signal, such as commit or rollback. Retries occur after a transient exception from a resource manager or remote partner, or if the configured asynchronous response timeout expires before all Web Services Atomic Transaction (WS-AT) partners have responded.
heuristicRetryWait	Specifies the number of seconds that the application server waits before retrying a completion signal, such as commit or rollback, after a transient exception from a resource manager or remote partner.
propogateOrBMTTranLifetimeTimeout	Specifies the number of seconds that a transaction remains inactive before it is rolled back.
asyncResponseTimeout	Specifies the amount of time, in seconds, that the server waits for an inbound Web Services Atomic Transaction (WS-AT) protocol response before resending the previous WS-AT protocol message.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["LPSHeuristicCompletion", "ROLLBACK"], ["WSTransactionSpecificationLevel", "WSTX_10"], ["enable", "true"]]`

Syntax

```
AdminServerManagement.configureTransactionService(nodeName, serverName, totalTranLifetimeTimeout, clientInactivityTimeout,
   maximumTransactionTimeout, heuristicRetryLimit, heuristicRetryWait,
   propogateOrBMTTranLifetimeTimeout, asyncResponseTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureTransactionService("myNode", "myServer",
    120, 60, 5, 2, 5, 300, 30,
    [["LPSHeuristicCompletion", "ROLLBACK"], ["WSTransactionSpecificationLevel", "WSTX_10"], ["enable", "true"]])
```

setJVMProperties

This script sets additional properties for your JVM configuration.

To run the script, specify the node name, server name, classpath, boot class path, initial heap size, maximum heap size, whether to enable debug mode, and debug arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.

Argument	Description
serverName	Specifies the name of the server of interest.
classPath	Optionally specifies the standard class path in which the Java virtual machine code looks for classes.
bootClasspath	Optionally specifies bootstrap classes and resources for JVM code. This option is only available for JVM instructions that support bootstrap classes and resources.
initialHeapSize	Optionally specifies the initial heap size available to the JVM code, in megabytes. Increasing the minimum heap size can improve startup. The number of garbage collection occurrences are reduced and a 10% gain in performance is realized. Increasing the size of the Java heap improves throughput until the heap no longer resides in physical memory, in general. After the heap begins swapping to disk, Java performance suffers drastically.
maxHeapSize	Optionally specifies the maximum heap size available to the JVM code, in megabytes. Increasing the heap size can improve startup. By increasing heap size, you can reduce the number of garbage collection occurrences with a 10% gain in performance.
debugMode	Optionally specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. If you set the debugMode argument to `true`, then you must specify debug arguments.
debugArgs	Optionally specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.

Syntax

```
AdminServerManagement.setJVMProperties(nodeName, serverName, classPath, bootClasspath, initialHeapSize, maxHeapSize, debugMode, debugArgs)
```

Example usage

```
AdminServerManagement.setJVMProperties("myNode", "myServer", "/a.jar", "", "", "", "", "")
```

setTraceSpecification

This script sets the trace specification for your configuration.

To run the script, specify the node name, server name, and trace specification arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
traceSpecification	Optionally specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.

Syntax

```
AdminServerManagement.setJVMProperties(nodeName, serverName, traceSpecification)
```

Example usage

```
AdminServerManagement.setTraceSpecification("myNode", "myServer", "com.ibm.ws.management.*=all")
```

configureCookieForServer

This script configures cookies in your application server configuration. Configure cookies to track sessions.

To run the script, specify the node name, server name, cookie name, domain, maximum cookie age, and whether to secure the cookie, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
cookieName	Specifies a unique name for the session management cookie. The servlet specification requires the name `JSESSIONID`. However, for flexibility this value can be configured.
domain	Specifies the domain field of a session tracking cookie. This value controls whether or not a browser sends a cookie to particular servers. For example, if you specify a particular domain, session cookies are sent to hosts in that domain. The default domain is the server.
maximumAge	Specifies the amount of time that the cookie lives on the client browser. Specify that the cookie lives only as long as the current browser session, or to a maximum age. If you choose the maximum age option, specify the age in seconds. This value corresponds to the Time to Live (TTL) value described in the Cookie specification. Default is the current browser session which is equivalent to setting the value to `-1`.
secure	Specifies that the session cookies include the secure field. Enabling the feature restricts the exchange of cookies to HTTPS sessions only.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["path", "C:/temp/mycookie"]]`

Syntax

```
AdminServerManagement.configureCookieForServer(nodeName, serverName, cookieName, domain, maximumAge, secure, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCookieForServer("myNode", "myServer", "myCookie", "uk.kingdom.com", -1, "true", [["path", "C:/temp/mycookie"]])
```

configureHTTPTransportForWebContainer

This script configures HTTP transports for a Web container. Transports provide request queues between application server plug-ins for Web servers and Web containers in which the Web modules of applications reside. When you request an application in a Web browser, the request is passed to the Web server, then along the transport to the Web container.

To run the script, specify the node name, server name, whether to adjust the port, whether external, the Secure Socket Layer (SSL) configuration to use, and whether to enable SSL, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
adjustPort	Specifies whether to automatically adjust the port for the Web container of interest.
external	Specifies whether to set the HTTP Transport for the Web container to external.
sslConfig	Specifies the Secure Sockets Layer (SSL) settings type for connections between the WebSphere Application Server plug-in and application server. The options include one or more SSL settings defined in the Security Center; for example, DefaultSSLSettings, ORBSSLSettings, or LDAPSSLSettings.
sslEnabled	Specifies whether to protect connections between the WebSphere Application Server plug-in and application server with Secure Sockets Layer (SSL). The default is not to use SSL.

Syntax

```
AdminServerManagement.configureHTTPTransportForWebContainer(nodeName, serverName, adjustPort, external, sslConfig, sslEnabled)
```

Example usage

```
AdminServerManagement.configureHTTPTransportForWebContainer("myNode", "myServer", "true", "true", "mySSLConfig", "true")
```

configureSessionManagerForServer

This script configures the session manager for the application server. Sessions allow applications running in a Web container to keep track of individual users.

To run the script, specify the node name, server name, and session persistence mode, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
sessionPersistenceMode	Specifies the session persistence mode. Valid values include `DATABASE`, `DATA_REPLICATION`, and `NONE`.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["accessSessionOnTimeout", "true"], ["enabled", "true"]]`

Syntax

```
AdminServerManagement.configureSessionManagerForServer(nodeName, serverName, sessionPersistenceMode, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureSessionManagerForServer("myNode", "myServer", "DATABASE", [["accessSessionOnTimeout", "true"], ["enabled", "true"]])
```

configureWebContainer

This script configures Web containers in your application server configuration. A Web container handles requests for servlets, JavaServer Pages (JSP) files, and other types of files that include server-side code. The Web container creates servlet instances, loads and unloads servlets, creates and manages request and response objects, and performs other servlet management tasks.

To run the script, specify the node name, server name, default virtual host name, and whether to enable servlet cache, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
webContainerName	Specifies the name of the Web container of interest.
defaultVirtualHostName	Specifies a virtual host that enables a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine. Valid values include: **default_host** The product provides a default virtual host with some common aliases such as the machine IP address, short host name, and fully qualified host name. The alias comprises the first part of the path for accessing a resource such as a servlet. For example, it is `localhost:9080` in the request `http://localhost:9080/myServlet`. **admin_host** This is another name for the application server; also known as server1 in the base installation. This process supports the use of the administrative console. **proxy_host** The virtual host called proxy_host, includes default port definitions, port 80 and 443, which are typically initialized as part of the proxy server initialization. Use this proxy host as appropriate with routing rules associated with the proxy server.
enableServletCaching	Specifies that if a servlet is invoked once and it generates output to be cached, a cache entry is created containing not only the output, but also side effects of the invocation. These side effects can include calls to other servlets or JavaServer Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information. Portlet fragment caching requires that servlet caching is enabled. Therefore, enabling portlet fragment caching automatically enables servlet caching. Disabling servlet caching automatically disables portlet fragment caching.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["allowAsyncRequestDispatching", "true"], ["disablePooling", "true"], ["sessionAffinityTimeout", 20]]`

Syntax

```
AdminServerManagement.configureWebContainer(nodeName, serverName, defaultVirtualHostName, enableServletCaching, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureWebContainer("myNode", "myServer", "myVH.uk.kingdom.com", true,
   [["allowAsyncRequestDispatching", "true"], ["disablePooling", "true"], ["sessionAffinityTimeout", 20]])
```

configureJavaProcessLogs

This script configures Java process logs for the application server. The system creates the JVM logs by redirecting the System.out and System.err streams of the JVM to independent log files.

To run the script, specify the Java process definition of interest and root directory for the process logs, as defined in the following table:

Argument	Description
javaProcessDefConfigID	Specifies the configuration ID of the Java Process Definition of interest.
processLogRoot	Specifies the root directory for the process logs.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["stdinFilename", "/temp/mystdin.log"]]`

Syntax

```
AdminServerManagement.configureJavaProcessLogs(javaProcessDefConfigID, processLogRoot, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureJavaProcessLogs
   ("(cells/WAS00Network/nodes/ndnode1/servers/server1|server.xml#JavaProcessDef_1184194176408)",
    "/temp/myJavaLog", [["stdinFilename", "/temp/mystdin.log"]])
```

configurePerformanceMonitoringService

This script configures performance monitoring infrastructure (PMI) in your configuration. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.

To run the script, specify the node name, server name, whether to enable PMI, and the initial specification level arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
enable	Specifies whether the application server attempts to enable Performance Monitoring Infrastructure (PMI). If an application server is started when the PMI is disabled, you have to restart the server in order to enable it.

Argument	Description
initialSpecLevel	Specifies a pre-defined set of Performance Monitoring Infrastructure (PMI) statistics for all components in the server. **None** All statistics are disabled. **Basic** Provides basic monitoring for application server resources and applications. This includes Java Platform Enterprise Edition (Java EE) components, HTTP session information, CPU usage information, and the top 38 statistics. This is the default setting. **Extended** Provides extended monitoring, including the basic level of monitoring plus workload monitor, performance advisor, and Tivoli resource models. Extended provides key statistics from frequently used WebSphere Application Server components. **All** Enables all statistics. **Custom** Provides fine-grained control with the ability to enable and disable individual statistics.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["statisticSet", "test statistic set"], ["synchronizedUpdate", "true"]]`

Syntax

```
AdminServerManagement.configurePerformanceMonitoringService(nodeName, serverName, enable, initialSpecLevel, otherAttributeList)
```

Example usage

```
AdminServerManagement.configurePerformanceMonitoringService("myNode", "myServer", "true", "Basic",
[["statisticSet", "test statistic set"], ["synchronizedUpdate", "true"]])
```

configurePMIRequestMetrics

This script configures PMI request metrics in your configuration. Request metrics provide data about each transaction, correlating this information across the various product components to provide an end-to-end picture of the transaction. To run the script, specify whether to enable request metrics and the trace level, as defined in the following table:

Argument	Description
enable	Specifies whether to turn on the request metrics feature. When disabled, the request metrics function is disabled.
traceLevel	Specifies how much trace data to accumulate for a given transaction. Note that trace level and components to be instrumented work together to control whether or not a request will be instrumented. **NONE** No instrumentation. **HOPS** Generates instrumentation information on process boundaries only (for example, a servlet request coming from a browser or a Web server and a JDBC request going to a database). **PERFORMANCE_DEBUG** Generates the data at Hops level and the first level of the intra-process servlet and Enterprise JavaBeans (EJB) call (for example, when an inbound servlet forwards to a servlet and an inbound EJB calls another EJB). Other intra-process calls like naming and service integration bus (SIB) are not enabled at this level. **DEBUG** Provides detailed instrumentation data, including response times for all intra-process calls. Requests to servlet filters will only be instrumented at this level.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["armType", "TIVOLI_ARM"], ["enableARM", "true"]]`

Syntax

```
AdminServerManagement.configurePMIRequestMetrics(enable, traceLevel, otherAttributeList)
```

Example usage

```
AdminServerManagement.configurePMIRequestMetrics("true", "DEBUG",
   [["armType", "TIVOLI_ARM"], ["enableARM", "true"]])
```

configureServerLogs

This script configures server logs for the application server of interest. To run the script, specify the node name, server name, and root directory for the server logs, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
serverLogRoot	Specifies the root directory for the server logs.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["formatWrites", "true"], ["messageFormatKind", "BASIC"], ["rolloverType", "BOTH"]]`

Syntax

```
AdminServerManagement.configureServerLogs(nodeName, serverName, serverLogRoot, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureServerLogs("myNode", "myServer", "/temp/mylog",
   [["formatWrites", "true"], ["messageFormatKind", "BASIC"], ["rolloverType", "BOTH"]])
```

configureTraceService

This script configures trace settings for the application server. Configure trace to obtain detailed information about running the application server. To run the script, specify the node name, server name, trace specification, and output type arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
startupTraceSpecification	Specifies the trace specification to enable for the component of interest. For example, the `com.ibm.ws.webservices.trace.MessageTrace=all` trace specification traces the contents of a SOAP message, including the binary attachment data.
traceOutputType	Specifies where trace output should be written. The trace output can be written directly to an output file, or stored in memory.
otherAttributeList	Optionally specifies additional attributes for the trace service using the following name and value pair format: `[["enable", "true"], ["traceFormat", "LOG_ANALYZER"]]`

Syntax

```
AdminServerManagement.configureTraceService(nodeName, serverName, traceString, outputType, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureTraceService("myNode", "myServer", "com.ibm.ws.management.*=all=enabled",
   "SPECIFIED_FILE", [["enable", "true"], ["traceFormat", "LOG_ANALYZER"]])
```

Server configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to create application servers, Web servers, and generic servers. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V70 directory. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: `""`.

Use the following script procedures to administer your application server:

- “createApplicationServer”
- “createAppServerTemplate”
- “createGenericServer”
- “createWebServer” on page 102
- “deleteServer” on page 103
- “deleteServerTemplate” on page 103

createApplicationServer

This script creates a new application server in your environment. During the installation process, the product creates a default application server, named `server1`. Most installations require several application servers to handle the application serving needs of their production environment.

To run the script, specify the node, server, and template names, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node on which to create the application server.
serverName	Specifies the name of the server to create.
templateName	Optionally specifies the template to use to create the application server.

Syntax

```
AdminServerManagement.createApplicationServer(nodeName, serverName, templateName)
```

Example usage

```
AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
```

createAppServerTemplate

This script creates a new application server template in your configuration. A server template is used to define the configuration settings for a new application server. When you create a new application server, you either select the default server template or a template you previously created, that is based on another, already existing application server. The default template is used if you do not specify a different template when you create the server.

To run the script, specify the node name, server name, and new template name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the node that corresponds to the server from which to base the template.
serverName	Specifies the name of the server from which to base the template.
newTemplateName	Specifies the name of the new template to create.

Syntax

```
AdminServerManagement.createAppServerTemplate(nodeName, serverName, newTemplateName)
```

Example usage

```
AdminServerManagement.createAppServerTemplate("myNode", "myServer", "myNewTemplate")
```

createGenericServer

This script configures a new generic server in the configuration. A generic server is a server that the application server manages, but does not supply. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: *""*.

To run the script, specify the node name, new server name, template name, start command path and arguments, working directory, and stop command path and arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node on which to create the server.
newServerName	Specifies the name of the server to create.
templateName	Optionally specifies the template to use to create the server.
startCmdPath	Optionally specifies the path to the command that will run when this generic server is started.
startCmdArguments	Optionally specifies the arguments to pass to the startCommand when the generic server is started.
workingDirectory	Optionally specifies the working directory for the generic server.
stopCmdPath	Optionally specifies the path to the command that will run when this generic server is stopped.
stopCmdArguments	Optionally specifies the arguments to pass to the stopCommand parameter when the generic server is stopped.

Syntax

```
AdminServerManagement.createGenericServer(nodeName, newServerName, templateName,
startCmdPath, startCmdArguments, workingDirectory, stopCmdPath,
 stopCmdArguments)
```

Example usage

```
AdminServerManagement.createGenericServer("myNode", "myServer",
 "default", "", "", "/temp", "", "")
```

createWebServer

This script configures a Web server in your configuration. An application server works with a Web server to handle requests for dynamic content, such as servlets, from Web applications. A Web server uses Web server plug-ins to establish and maintain persistent HTTP and HTTPS connections with an application server. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: *""*.

To run the script, specify the node name, new server name, port number, server install root, plug-in installation root, configuration file path, Windows Operating System service name, error log path, access log path, and web protocol type, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node on which the Web server is defined.
newServerName	Specifies the name of the Web server to create.
port	Optionally specifies the port from which to ping the status of the Web server.
serverInstallRoot	Optionally specifies the fully qualified path where the Web server is installed. This field is required if you are using IBM® HTTP Server. For all other Web servers, this field is not required. If you enable any administrative function for non-IBM HTTP Server Web servers, the installation path is necessary.
pluginInstallPath	Specifies the installation path for the Web server plug-in.
configFilePath	Specifies the configuration file for your Web server. Specify the file and not just the directory of the file. The application server generates the plugin-cfg.xml file by default. The configuration file identifies applications, application servers, clusters, and HTTP ports for the Web server. The Web server uses the file to access deployed applications on various application servers.
errorLogPath	Specifies the location of the error log file.
accessLogPath	Specifies the location of the access log file.
webProtocol	Specifies the protocol to use for Web server communications. Use the `HTTPS` protocol to securely communicate with the Web server. The default is `HTTP`.

Syntax

```
AdminServerManagement.createWebServer(nodeName, newServerName, port,
 serverInstallRoot, pluginInstallPath, configFilePath,
 errorLogPath,
  accessLogPath, webProtocol)
```

Example usage

```
AdminServerManagement.createWebServer("myNode", "myWebServer", "", "", "", "", "", "", "", "" )
```

deleteServer

This script removes a server from the application server configuration.

To run the script, specify the node and server names, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server to delete.

Syntax

```
AdminServerManagement.deleteServer(nodeName, serverName)
```

Example usage

```
AdminServerManagement.deleteServer("myNode", "myServer")
```

deleteServerTemplate

This script deletes a server template from your configuration.

To run the script, specify the template name, as defined in the following table:

Argument	Description
templateName	Specifies the name of the template to delete.

Syntax

```
AdminServerManagement.deleteServerTemplate(templateName)
```

Example usage

```
AdminServerManagement.deleteServerTemplate("newServerTemplate")
```

Server query scripts

The scripting library provides multiple script procedures to automate your server configurations. This topic provides usage information for scripts that query your application server configuration. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V70 directory. Use the following script procedures to query your application server configuration:

checkIfServerExists

This script determines whether the server of interest exists in your configuration. To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.checkIfServerExists(nodeName, serverName)
```

Example usage

```
AdminServerManagement.checkIfServerExists("myNode", "myServer")
```

checkIfServerTemplateExists

This script determines whether the server template of interest exists in your configuration. To run the script, specify the template name arguments, as defined in the following table:

Argument	Description
templateName	Specifies the name of the server template of interest.

Syntax

```
AdminServerManagement.checkIfServerTemplateExists(templateName)
```

Example usage

```
AdminServerManagement.checkIfServerTemplateExists("newServer")
```

getJavaHome

This script displays the Java home value. To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getJavaHome(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getJavaHome("myNode", "myServer")
```

getServerProcessType

This script displays the type of server process for a specific server. To run the script, specify the node and server name arguments for the server of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getServerProcessType(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getServerProcessType("myNode", "server1")
```

getServerPID

This script displays the running server process ID for a specific target. To run the script, specify the node and server name arguments for the server of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getServerPID(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getServerPID("myNode", "server1")
```

help

This script displays the script procedures that the AdminServerManagement script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
scriptName	Specifies the name of the script of interest.

Syntax

```
AdminServerManagement.help(scriptName)
```

Example usage

```
AdminServerManagement.help("getServerProcessType")
```

listJVMProperties

This script displays the properties that are associated with your Java virtual machine (JVM) configuration. To run the script, specify the node name, server name, and optionally the JVM property of interest, as defined in the following table:

Argument	Description
nodeName	Optionally specifies the name of the node of interest.
serverName	Optionally specifies the name of the server of interest.
JVMProperty	Optionally specifies the JVM property to query.

Syntax

```
AdminServerManagement.listJVMProperties(nodeName, serverName, JVMProperty)
```

Example usage

```
AdminServerManagement.listJVMProperties("myNode", "myServer", "")
```

listServers

This script displays the servers that exist in your configuration. You can optionally specify the node name or server type to query for a specific scope, as defined in the following table:

Argument	Description
serverType	Specifies the name of the server to query.
nodeName	Specifies the name of the node to query.

Syntax

```
AdminServerManagement.listServers(serverType, nodeName)
```

Example usage

```
AdminServerManagement.listServers("APPLICATION_SERVER", "myNode")
```

listServerTemplates

This script displays the server templates in your configuration. To run the script, specify the template version, server type, and template name, as defined in the following table:

Argument	Description
templateVersion	Optionally specifies the version of the template of interest.
serverType	Optionally specifies the type of server. Valid values include the `GENERIC_SERVER`, `WEB_SERVER`, `APPLICATION_SERVER` , and `PROXY_SERVER` server types.
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminServerManagement.listServerTemplates(templateVersion, serverType, templateName)
```

Example usage

```
AdminServerManagement.listServerTemplates("", "APPLICATION_SERVER", "default")
```

listServerTypes

This script displays the server types that are available on the node of interest. To run the script, specify the node name, as defined in the following table:

Argument	Description
nodeName	Optionally specifies the name of the node of interest.

Syntax

```
AdminServerManagement.listServerTypes(nodeName)
```

Example usage

```
AdminServerManagement.listServerTypes("myNode")
```

queryingMBeans

This script queries the application server for Managed Beans (MBeans). Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mbeanType	Specifies the type of MBean to query.

Syntax

```
AdminServerManagement.queryingMBeans(nodeName, serverName, mbeanType)
```

Example usage

```
AdminServerManagement.queryingMBeans("myNode", "server1", "Server")
```

showServerInfo

This script displays server configuration properties for the server of interest. The script displays the cell name, server type, product version, node name, and server name. To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.showServerInfo(nodeName, serverName)
```

Example usage

```
AdminServerManagement.showServerInfo("myNode", "myServer")
```

viewingProductInformation

This script displays the application server product version.

Syntax

```
AdminServerManagement.viewingProductInformation()
```

Example usage

```
AdminServerManagement.viewingProductInformation()
```

Server administration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to delete, start, and stop servers. You can run each script individually or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V61 directory.

Use the following script procedures to administer your application server:

- “startAllServers” on page 108
- “startSingleServer” on page 108
- “stopAllServers” on page 108
- “stopSingleServer” on page 108

startAllServers

This script starts all servers on a node in your configuration.

To run the script, specify the node name, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.

Syntax

```
AdminServerManagement.startAllServers(nodeName)
```

Example usage

```
AdminServerManagement.startAllServers("myNode")
```

startSingleServer

This script starts a specific server in your configuration.

To run the script, specify the node name and server name, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server to start.

Syntax

```
AdminServerManagement.startSingleServer(nodeName, serverName)
```

Example usage

```
AdminServerManagement.startSingleServer("myNode", "myServer")
```

stopAllServers

This script stops all servers on a node in your configuration.

To run the script, specify the node name, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.

Syntax

```
AdminServerManagement.stopAllServers(nodeName)
```

Example usage

```
AdminServerManagement.stopAllServers("myNode")
```

stopSingleServer

This script stops a single server in your configuration.

To run the script, specify the node name and server name, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.stopSingleServer(nodeName, serverName, classname, displayname, classpath, otherAttributeList)
```

Example usage

```
AdminServerManagement.stopSingleServer("myNode", "myServer")
```

Automating administrative architecture setup using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the server, node, and cluster management scripts to configure servers, nodes, node groups, and clusters in your application server environment.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Use the scripts in the following directories to configure your administrative architecture:

- The server and cluster management procedures are located in the *app_server_root*/scriptLibraries/servers/V70 subdirectory.
- The node and node group management procedures are located in the *app_server_root*/scriptLibraries/system/V70 subdirectory.

Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory, and save existing automation scripts in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

This topic provides one sample combination of procedures. Use the following steps to create a node group and add three nodes to the group:

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Display the nodes in your environment.

   Run the listNodes script procedure from the AdminNodeManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminNodeManagement.listNodes()"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeManagement.listNodes()
```

   For this example, the command returns the following output:

```
Node1
Node2
Node3
```

3. Create a node group.

   Run the createNodeGroup script procedure from the AdminNodeManagement script library, specifying the name to assign to the new node group, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminNodeManagement.createNodeGroup("NodeGroup1")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeManagement.createNodeGroup("myNodeGroup")
```

4. Add nodes to the node group.

   Run the addNodeGroupMember script procedure from the AdminNodeManagement script library to add the `Node1`, `Node2`, and `Node3` nodes to the `NodeGroup1` node group, specifying the node name and node group name, as the following examples demonstrate:

```
wsadmin -lang jython -c "AdminNodeManagement.addNodeGroupMember("Node1", "NodeGroup1")"
wsadmin -lang jython -c "AdminNodeManagement.addNodeGroupMember("Node2", "NodeGroup1")"
wsadmin -lang jython -c "AdminNodeManagement.addNodeGroupMember("Node3", "NodeGroup1")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeManagement.addNodeGroupMember("Node1", "NodeGroup1")
wsadmin>AdminNodeManagement.addNodeGroupMember("Node2", "NodeGroup1")
wsadmin>AdminNodeManagement.addNodeGroupMember("Node3", "NodeGroup1")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Node administration scripts

The scripting library provides multiple script procedures to automate your server configurations. This topic provides usage information for scripts that query, configure, and manage your node configurations. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All node management script procedures are located in the *app_server_root*/scriptLibraries/system/V70 directory. Use the following script procedures to query, configure, and manage your node configurations:

- "configureDiscoveryProtocolOnNode"
- "doesNodeExist" on page 112
- "isNodeRunning" on page 112
- "listNodes" on page 112
- "restartActiveNodes" on page 112
- "restartNodeAgent" on page 113
- "stopNode" on page 113
- "stopNodeAgent" on page 113
- "syncActiveNodes" on page 114
- "syncNode" on page 114

configureDiscoveryProtocolOnNode

This script configures the discovery protocol for the node of interest. If the discovery protocol that a node uses is not appropriate for the node, modify the configuration to use the appropriate protocol.

To run the script, specify the node of interest and the protocol, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.
discoveryProtocol	Specifies the protocol that the node follows to retrieve information from a network. The Discovery protocol setting is only valid for managed nodes. Specify Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). UDP is faster than TCP, but TCP is more reliable than UDP because UDP does not guarantee delivery of datagrams to the destination. Between these two protocols, the TCP default is recommended.

Syntax

```
AdminNodeManagement.configureDiscoveryProtocolOnNode(nodeName, discoveryProtocol)
```

Example usage

```
AdminNodeManagement.configureDiscoveryProtocolOnNode("myNode", "UDP")
```

doesNodeExist

This script displays a value of 1 if the node of interest exists, or a value of -1 if the node of interest does not exist. To run the script, specify the name of the node, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node to query. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.doesNodeExist("nodeName")
```

Example usage

```
AdminNodeManagement.doesNodeExist("myNode")
```

isNodeRunning

This script displays a value of 1 if the node of interest can be started, or a value of -1 if the node of interest cannot be started. To run the script, specify the name of the node, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.isNodeRunning(nodeName)
```

Example usage

```
AdminNodeManagement.isNodeRunning("myNode")
```

listNodes

This script displays a list of nodes in your environment.

Syntax

```
AdminNodeManagement.listNodes()
```

Example usage

```
AdminNodeManagement.listNodes()
```

restartActiveNodes

This script restarts the nodes in your environment with node agents that are in the started state.

Syntax

```
AdminNodeManagement.restartActiveNodes()
```

Example usage

```
AdminNodeManagement.restartActiveNodes()
```

restartNodeAgent

This script restarts the node agent of interest. Node agents are administrative agents that monitor application servers on a host system and route administrative requests to servers. A node agent is the running server that represents a node in a Network Deployment environment.

To run the script, specify the node of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node to restart. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.restartNodeAgent(nodeName)
```

Example usage

```
AdminNodeManagement.restartNodeAgent("myNode")
```

stopNode

This script stops the node of interest. Start or stop a node as needed when administering your Network Deployment environment. Before your environment can service requests, you must have the deployment manager and node started, and typically an HTTP server running.

To run the script, specify the node of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node to stop. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.stopNode(nodeName)
```

Example usage

```
AdminNodeManagement.stopNode("myNode")
```

stopNodeAgent

This script stops the node agent of interest. Node agents are administrative agents that monitor application servers on a host system and route administrative requests to servers. A node agent is the running server that represents a node in a Network Deployment environment.

To run the script, specify the node of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.stopNodeAgent(nodeName)
```

Example usage

```
AdminNodeManagement.stopNodeAgent("myNode")
```

syncActiveNodes

This script propagates configuration changes to each active node in your environment. By default, this situation occurs periodically, as long as the node can communicate with the deployment manager.

Syntax

```
AdminNodeManagement.syncActiveNodes()
```

Example usage

```
AdminNodeManagement.syncActiveNodes()
```

syncNode

This script propagates configuration changes to the node of interest. By default, this situation occurs periodically, as long as the node can communicate with the deployment manager.

To run the script, specify the node of interest, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node. The node name is unique within the cell. A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

Syntax

```
AdminNodeManagement.syncNode(nodeName)
```

Example usage

```
AdminNodeManagement.syncNode("myNode")
```

Node group configuration scripts

The scripting library provides multiple script procedures to automate your server configurations. Use the information in this topic to use scripts that query, configure, and manage your node configurations. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

Use node groups to define groups of nodes can host members of the same cluster. An application that is deployed to a cluster must be capable of running on any of the cluster members. The node that hosts each of the cluster members must be configured with software and settings that are necessary to support the application.

By organizing nodes that satisfy your application requirements into a node group, you establish an administrative policy that governs which nodes can be used together to form a cluster. Those who define the cell configuration and those who create server clusters can operate with more independence from one another.

All node management script procedures are located in the *app_server_root*/scriptLibraries/system/V70 directory. Use the following script procedures to query, configure, and manage your node configurations:

- “addNodeGroupMember” on page 115
- “checkIfNodeExists” on page 115
- “checkIfNodeGroupExists” on page 115
- “createNodeGroup” on page 116

addNodeGroupMember

This script adds a node to a node group that exists in your configuration.

To run the script, specify the name of the node and the node group, as defined in the following table:

Argument	Description
nodeName	Specifies a logical name for the node group member. A node group member is a node. The name must be unique within the cell. A node group member name typically is identical to the host name for the computer.
nodeGroupName	Specifies a logical name for the node group. The name must be unique within the cell. The name can start with a number.

Syntax

```
AdminNodeManagement.addNodeGroupMember(nodeName, discoveryProtocol)
```

Example usage

```
AdminNodeManagement.addNodeGroupMember("myNode", "myNodeGroup")
```

checkIfNodeExists

This script displays whether the node of interest exists in a specific node group.

To run the script, specify the node group and node arguments, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group to query.
nodeName	Specifies the name of the node to query.

Syntax

```
AdminNodeManagement.checkIfNodeExists(nodeGroupName, nodeName)
```

Example usage

```
AdminNodeManagement.checkIfNodeExists("myNodeGroup", "myNode")
```

checkIfNodeGroupExists

This script displays whether a specific node group exists in your configuration.

To run the script, specify the name of the node group, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group to query.

Syntax

```
AdminNodeManagement.checkIfNodeGroupExists(nodeGroupName)
```

Example usage

```
AdminNodeManagement.checkIfNodeGroupExists("myNodeGroup")
```

createNodeGroup

This script creates a new node group in your configuration.

To run the script, specify the name of the node group, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group to create.

Syntax

```
AdminNodeManagement.createNodeGroup("nodeGroupName")
```

Example usage

```
AdminNodeManagement.createNodeGroup("myNodeGroup")
```

createNodeGroupProperty

This script assigns custom properties to the node group of interest.

To run the script, specify the name of the node, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node of interest.
customPropertyName	Specifies the name, or key, for the property. Each property name must be unique. If the same name is used for multiple properties, the value specified for the first property that has that name is used. Do not start your property names with `was.` because this prefix is reserved for properties that are predefined in the application server.
customPropertyValue	Specifies the value to assign to the custom property name.
customPropertyDesc	Optionally specifies a description for the custom property to create.
isPropertyRequired	Optionally specifies whether the custom property is required in your configuration. Specify `true` to set the custom property as required in your configuration.

Syntax

```
AdminNodeManagement.createNodeGroupProperty(nodeGroupName, customPropertyName, customPropertyValue, customPropertyDesc, isPropertyRequired)
```

Example usage

```
AdminNodeGroupManagement.createNodeGroupProperty("myNodeGroup", "myProp", "myPropValue", "this is my prop", "true")
```

deleteNodeGroup

This script deletes a node group from your configuration.

To run the script, specify the node group name, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group to delete.

Syntax

```
AdminNodeManagement.deleteNodeGroup(nodeGroupName)
```

Example usage

```
AdminNodeManagement.deleteNodeGroup("myNodeGroup")
```

deleteNodeGroupMember

This script removes a node from a specific node group in your configuration.

To run the script, specify the node group name and node name arguments, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.
nodeName	Specifies the name of the node to remove from the node group.

Syntax

```
AdminNodeManagement.deleteNodeGroupMember(nodeGroupName, nodeName)
```

Example usage

```
AdminNodeManagement.deleteNodeGroupMember("myNodeGroup", "myNode")
```

deleteNodeGroupProperty

This script removes a specific custom property from a node group.

To run the script, specify the node group name and property name arguments, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.
customPropertyName	Specifies the name of the custom property to remove from your node group configuration.

Syntax

```
AdminNodeManagement.deleteNodeGroupProperty(nodeGroupName, customPropertyName)
```

Example usage

```
AdminNodeManagement.deleteNodeGroupProperty("myNodeGroup", "myProp")
```

help

This script displays the script procedures that the AdminNodeGroupManagement script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminServerManagement.help(script)
```

Example usage

```
AdminServerManagement.help("modifyNodeGroupProperty")
```

listNodeGroups

This script displays the node groups that exist in your configuration. If you specify the name of a specific node, the script returns the name of the node group to which the node belongs.

Argument	Description
nodeName	Optionally specifies the name of the node to use to query the node groups.

Syntax

```
AdminNodeManagement.listNodeGroups()
```

Example usage

```
AdminNodeManagement.listNodeGroups()
```

listNodeGroupMembers

This script lists the name of each node that is configured within a specific node group.

To run the script, specify the node group argument, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.

Syntax

```
AdminNodeManagement.listNodeGroupMembers(nodeGroupName)
```

Example usage

```
AdminNodeManagement.listNodeGroupMembers("myNodeGroup")
```

listNodeGroupProperties

This script lists the custom properties that are configured within a specific node group.

To run the script, specify the node group argument, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.

Syntax

```
AdminNodeManagement.listNodeGroupProperties(nodeGroupName)
```

Example usage

```
AdminNodeManagement.listNodeGroupProperties("myNodeGroup")
```

modifyNodeGroup

This script modifies the short name and description of a node group.

To run the script, specify the node group, short name and description arguments, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.
shortName	Specifies the short name of the node group of interest.
description	Specifies a description of the node group.

Syntax

```
AdminNodeManagement.listNodeGroupProperties(nodeGroupName, shortName, description)
```

Example usage

```
AdminNodeManagement.listNodeGroupProperties("myNodeGroup", "NG1", "my first node group")
```

modifyNodeGroupProperty

This script modifies the value of a custom property assigned to a node group.

To run the script, specify the node group, custom property, custom property value, custom property description, and whether the property is required, as defined in the following table:

Argument	Description
nodeGroupName	Specifies the name of the node group of interest.
customPropertyName	Specifies the name of the custom property to modify.
customPropertyValue	Optionally specifies a new value for the custom property of interest.
customPropertyDescription	Optionally specifies a description for the custom property.
isPropertyRequired	Optionally specifies whether the custom property is required.

Syntax

```
AdminNodeManagement.modifyNodeGroupProperty(nodeGroupName, customPropertyName, customPropertyValue, customPropertyDescription, isPropertyRequired)
```

Example usage

```
AdminNodeManagement.modifyNodeGroupProperty("myNodeGroup", "customProp", "newValue", "new description of property", "false")
```

Cluster configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to configure clusters with or without cluster members, using a template, and to remove clusters from your configuration. You can run each script individually, or combine procedures to create custom automation scripts.

The AdminClusterManagement script procedures are located in the *app_server_root*/scriptLibraries/server/V70 directory.

Use the following script procedures to configure clusters in your environment:

- “createClusterMember”
- “createClusterWithFirstMember” on page 120
- “createClusterWithoutMember” on page 120
- “createFirstClusterMemberWithTemplate” on page 121
- “createFirstClusterMemberWithTemplateNodeServer” on page 121

Use the following script procedures remove clusters and cluster members from your configuration:

- “deleteCluster” on page 121
- “deleteClusterMember” on page 122

createClusterMember

This script assigns a server cluster member to a specific cluster. When you create the first cluster member, a copy of that member is stored as part of the cluster data and becomes the template for all additional cluster members that you create.

To run the script, specify the cluster name, node name, and new member name arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to which the system adds the cluster member.
nodeName	Specifies the name of the node on which the application server resides.
newMemberName	Specifies the name to assign to the cluster member.

Syntax

```
AdminClusterManagement.createClusterMember(clusterName, nodeName, newMemberName)
```

Example usage

```
AdminClusterManagement.createClusterMember("myCluster", "myNode", "clusterMember1")
```

createClusterWithFirstMember

This script creates a new cluster configuration and adds the first cluster member to the cluster. Use clusters to manage a group of application servers as a single unit, and distribute client requests among the application servers that are members of the cluster. Create a cluster to balance your client requests across multiple application servers and to provide a highly available environment for your applications.

To run the script, specify the cluster name, cluster type, node name, and server name arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name to assign to the new cluster.
clusterType	Specifies the type of cluster to create. You can specify a value of `APPLICATION_SERVER`, `GENERIC_SERVER`, or `WEB_SERVER`.
nodeName	Specifies the name of the node on which the cluster resides.
serverName	Specifies the name of the server to add to the cluster.

Syntax

```
AdminClusterManagement.createClusterWithFirstMember(clusterName, clusterType, nodeName, serverName)
```

Example usage

```
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "myServer")
```

createClusterWithoutMember

This script creates a new cluster configuration in your environment. Use clusters to manage a group of application servers as a single unit, and distribute client requests among the application servers that are members of the cluster. Create a cluster to balance your client requests across multiple application servers and to provide a highly available environment for your applications.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name to assign to the new cluster.

Syntax

```
AdminClusterManagement.createClusterWithoutMember(clusterName)
```

Example usage

```
AdminClusterManagement.createClusterWithoutMember("myCluster")
```

createFirstClusterMemberWithTemplate

This script uses a template to add the first server cluster member to a specific cluster. A copy of the first cluster member that you create is stored in the cluster scope as a template. You can create the first cluster member using any existing server as a template or a default server template. You can also create a first cluster member when you create the cluster by converting a server to a cluster. When you create a first cluster member, the template of the cluster member is stored in the scope of the cluster. Additional cluster members are created using the cluster member template stored in the cluster scope

To run the script, specify the cluster name, node name, new member name, and template name arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster of interest.
nodeName	Specifies the name of the node on which the application server resides.
newMemberName	Specifies the name to assign to the cluster member.
templateName	Specifies the name of the template to use to create the cluster member.

Syntax

```
AdminClusterManagement.createFirstClusterMemberWithTemplate(clusterName, nodeName, newMemberName, templateName)
```

Example usage

```
AdminClusterManagement.createFirstClusterMemberWithTemplate("myCluster", "myNode", "myClusterMember", "default")
```

createFirstClusterMemberWithTemplateNodeServer

This script uses a node with an existing application server as a template to create a new cluster member in your configuration. When you create the first cluster member, a copy of that member is stored as part of the cluster data and becomes the template for all additional cluster members that you create.

To run the script, specify the cluster name, node name, new member name, template node name, and template server name arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to which the system adds the cluster member.
nodeName	Specifies the name of the node on which the application server resides.
newMemberName	Specifies the name to assign to the cluster member.
templateNodeName	Specifies the name of the node with an existing application server to use as the template when creating the new cluster member.
templateServerName	Specifies the name of the existing application server to use as the model when creating the new cluster member.

Syntax

```
AdminClusterManagement.createFirstClusterMemberWithTemplateNodeServer(clusterName, nodeName, newMemberName, newMemberName, templateNodeName, templateServerName)
```

Example usage

```
AdminClusterManagement.createFirstClusterMemberWithTemplateNodeServer("myCluster", "myNode", "newClusterMember", "myTemplateNode", "myTemplateServer")
```

deleteCluster

This script deletes the configuration of a server cluster. A server cluster consists of a group of application servers that are referred to as cluster members. The script deletes the server cluster and each of its cluster members.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to remove from your configuration.

Syntax

```
AdminClusterManagement.deleteCluster(clusterName)
```

Example usage

```
AdminClusterManagement.deleteCluster("myCluster")
```

deleteClusterMember

This script removes a cluster member from your cluster configuration. A cluster member is a server that belongs to a cluster.

To run the script, specify the cluster name, node name, and server cluster member arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster from which to remove the cluster member.
nodeName	Specifies the name of the node that is associated with the cluster member to delete.
clusterMemberName	Specifies the name of the cluster member to remove from your configuration.

Syntax

```
AdminClusterManagement.deleteClusterMember(clusterName, nodeName, clusterMemberName)
```

Example usage

```
AdminClusterManagement.deleteClusterMember("myCluster", "myNode", "clusterMember1")
```

Cluster query scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to determine if clusters and cluster members exist and to display the clusters and cluster members that are configured in your environment. You can run each script individually, or combine procedures to create custom automation scripts.

The AdminClusterManagement script procedures are located in the *app_server_root*/scriptLibraries/server/V70 directory.

Use the following script procedures to query your cluster configuration:

- “checkIfClusterExists”
- “checkIfClusterMemberExists” on page 123
- “help” on page 123
- “listClusters” on page 123
- “listClusterMembers” on page 123

checkIfClusterExists

This script displays whether the cluster of interest exists in your configuration.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster of interest.

Syntax

```
AdminClusterManagement.checkIfClusterExists(clusterName)
```

Example usage

```
AdminClusterManagement.checkIfClusterExists("myCluster")
```

checkIfClusterMemberExists

This script displays whether a specific cluster member exists in your cluster configuration.

To run the script, specify the cluster name and server cluster member arguments, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to query.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminClusterManagement.checkIfClusterMemberExists(clusterName, serverName)
```

Example usage

```
AdminClusterManagement.checkIfClusterMemberExists("myCluster", "myClusterMember")
```

help

This script displays the script procedures that the AdminClusterManagement script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminResources.help(script)
```

Example usage

```
AdminResources.help("createClusterWithoutMember")
```

listClusters

This script displays each cluster that exists in your configuration. This script does not require arguments.

Syntax

```
AdminClusterManagement.listClusters()
```

Example usage

```
AdminClusterManagement.listClusters()
```

listClusterMembers

This script displays the server cluster members that exist in a specific cluster configuration.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster of interest.

Syntax

```
AdminClusterManagement.listClusterMembers(clusterName)
```

Example usage

```
AdminClusterManagement.listClusterMembers("myCluster")
```

Cluster administration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to start and stop cluster processes with a variety of options. You can run each script individually or combine procedures to create custom automation scripts.

The AdminClusterManagement script procedures are located in the *app_server_root*/scriptLibraries/server/v61 directory.

Use the following script procedures to start cluster processes in your environment:

- "rippleStartAllClusters"
- "rippleStartSingleCluster"
- "startAllClusters" on page 125
- "startSingleCluster" on page 125

Use the following script procedures to stop cluster processes in your environment:

- "immediateStopAllRunningClusters" on page 125
- "immediateStopSingleCluster" on page 125
- "stopAllClusters" on page 126
- "stopSingleCluster" on page 126

rippleStartAllClusters

This script stops and restarts each cluster within a cell configuration.

Syntax

```
AdminClusterManagement.rippleStartAllClusters()
```

Example usage

```
AdminClusterManagement.rippleStartAllClusters()
```

rippleStartSingleCluster

This script stops and restarts the cluster members within a specific cluster configuration.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to stop and restart.

Syntax

```
AdminClusterManagement.rippleStartSingleCluster(clusterName)
```

Example usage

```
AdminClusterManagement.rippleStartSingleCluster("myCluster")
```

startAllClusters

This script starts each cluster within a cell configuration.

Syntax

```
AdminClusterManagement.startAllClusters()
```

Example usage

```
AdminClusterManagement.startAllClusters()
```

startSingleCluster

This script starts a specific cluster in your configuration.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster of interest.

Syntax

```
AdminClusterManagement.startSingleCluster(clusterName)
```

Example usage

```
AdminClusterManagement.startSingleCluster("myCluster")
```

immediateStopAllRunningClusters

This script stops the server cluster members for each active cluster within a specific cell. The server ignores any current or pending tasks. When the stop operation begins, the cluster state changes to partially stopped. After all servers stop, the cluster state becomes stopped.

Syntax

```
AdminClusterManagement.immediateStopAllRunningClusters()
```

Example usage

```
AdminClusterManagement.immediateStopAllRunningClusters()
```

immediateStopSingleCluster

This script stops the server cluster members for a specific cluster within a cell. The server ignores any current or pending tasks. When the stop operation begins, the cluster state changes to partially stopped. After all servers stop, the cluster state becomes stopped.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to stop.

Syntax

```
AdminClusterManagement.immediateStopSingleCluster(clusterName)
```

Example usage

```
AdminClusterManagement.immediateStopSingleCluster("myCluster")
```

stopAllClusters

This script stops the server cluster members of each active cluster within a specific cell. Each server stops so that the server can complete existing requests and allow failover to another member of the cluster. When the stop operation begins the cluster state changes to partially stopped. After all servers stop, the cluster state becomes stopped.

Syntax

```
AdminClusterManagement.stopAllClusters()
```

Example usage

```
AdminClusterManagement.stopAllClusters()
```

stopSingleCluster

This script stops the server cluster members of a specific active cluster within a cell. Each server stops so that the server can complete existing requests and allow failover to another member of the cluster. When the stop operation begins the cluster state changes to partially stopped. After all servers stop, the cluster state becomes stopped.

To run the script, specify the cluster name argument, as defined in the following table:

Argument	Description
clusterName	Specifies the name of the cluster to stop.

Syntax

```
AdminClusterManagement.stopSingleCluster(clusterName)
```

Example usage

```
AdminClusterManagement.stopSingleCluster("myCluster")
```

Automating application configurations using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage applications in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  ```

```
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
    "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The application management procedures in scripting library are located in the *app_server_root*/scriptLibraries/application/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminApplication.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. Use the following steps to use the scripting library to install an application on a cluster and start the application:

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create a cluster.

   Run the createClusterWithoutMember script procedure from the AdminClusterManagement script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterWithoutMember('myCluster')"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminClusterManagement.createClusterWithoutMember("myCluster")
```

3. Create a cluster member for the new cluster.

   Run the createClusterMember script procedure from the AdminClusterManagement script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterMember('myCluster', 'myNode, 'myNewMember')"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminClusterManagement.createClusterWithoutMember("myCluster", "myNode", "myNewMember")
```

4. Install the application on the newly created cluster.

Run the installAppWithClusterOption script procedure from the AdminApplication script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminApplication.installAppWithClusterOption('myApplication','myApplicationEar.ear','myCluster')"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminApplication.installAppWithClusterOption("myApplication", "myApplicationEar.ear", "myCluster")
```

5. Start the application on the cluster.

   Run the startApplicationOnCluster script procedure from the AdminApplication script library and specify the required arguments, as the following example displays:

```
bin>wsadmin -lang jython -c "AdminApplication.startApplicationOnCluster('myApplication','myCluster')"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication", "myCluster")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Application installation and uninstallation scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that install applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to install and uninstall applications:

installAppWithAppNameOption

This script installs an application using the -appname option. The -appname option specifies the name of the application. The default is the display name of the application.

To run the script, specify the application name and Enterprise Archive (EAR) file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.

Syntax

```
AdminApplication.installAppWithAppNameOption(appName,
 earFile)
```

Example usage

```
AdminApplication.installAppWithAppNameOption("myApp", "\ears\DefaultApplication.ear")
```

installAppWithDefaultBindingOption

This script installs an application using the -usedefaultbindings option.

To run the script, specify the application name, Enterprise Archive (EAR) file, data source Java Naming and Directory Interface (JNDI) name, data source user name, data source password, connection factory, Enterprise JavaBeans prefix, and virtual host name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the target node.
serverName	Specifies the name of the target server.
dsJndiName	Specifies the JNDI name of the data source to use.
dsUserName	Specifies the user name for the data source.
dsPassword	Specifies the password for the data source.
connFactory	Specifies the name of the connection factory to use.
EJBprefix	Specifies the Enterprise JavaBean (EJB) prefix to use.
virtualHostName	Specifies the virtual host for the application to install.

Syntax

```
AdminApplication.installAppWithDefaultBindingOption(appName,
 earFile, nodeName, serverName, dsJndiName,
 dsUserName, dsPassword, connFactory, EJBprefix,
 virtualHostName)
```

Example usage

```
AdminApplication.installAppWithDefaultBindingOption("myApp", "\ears\DefaultApplication.ear", "myNode",
 "myServer", "myJndi", "user1", "password", "myCf", "myEjb", "myVH")
```

installAppWithNodeAndServerOptions

This script installs an application using the -node and -server options.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppWithNodeAndServerOptions(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppWithNodeAndServerOptions("myApp", "/ears/DefaultApplication.ear",
 "myNode", "myServer")
```

installAppWithClusterOption

This script installs an application using the -cluster option.

To run the script, specify the application name, EAR file, and cluster name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
clusterName	Specifies the name of the cluster of interest.

Syntax

```
AdminApplication.installAppWithClusterOption(appName,
 earFile, clusterName)
```

Example usage

```
AdminApplication.installAppWithClusterOption("myApp", "/ears/DefaultApplication.ear",
 "myCluster")
```

installAppModulesToSameServerWithMapModulesToServersOption

This script deploys application modules to the same server using the -MapModulesToServers option.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppModulesToSameServerWithMapModulesToServersOption(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppModulesToSameServerWithMapModulesToServersOption("myApp",
 "/ears/DefaultApplication.ear", "myNode", "myServer")
```

installAppModulesToDiffServersWithMapModulesToServersOption

This script deploys application modules to different servers using the -MapModulesToServers option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and both server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToDiffServersWithMapModulesToServersOption(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToDiffServersWithMapModulesToServersOption("myApp",
 "/ears/DefaultApplication.ear", "myCell", "myNode", "myServer1", "myServer2")
```

installAppModulesToSameServerWithPatternMatching

This script deploys application modules with the -MapModulesToServers pattern matching option.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppModulesToSameServerWithPatternMatching(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppModulesToSameServerUingPatternMatching("myApp",
 "/ears/DefaultApplication.ear", "myNode", "myServer")
```

installAppModulesToDiffServersWithPatternMatching

This script deploys application modules to different servers using the -MapModulesToServers pattern matching option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and both server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToDiffServersWithPatternMatching(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToDiffServersWithPatternMatching("myApp", "/ears/DefaultApplication.ear", "myNode", "myServer1", "myServer2")
```

installAppModulesToMultiServersWithPatternMatching

This script deploys application modules to multiple servers using the -MapModulesToServers pattern matching option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and each server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToMultiServersWithPatternMatching(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToMultiServersWithPatternMatching("myApp",
 "/ears/DefaultApplication.ear", "myCell", "myNode", "myServer1", "myServer2")
```

installAppWithTargetOption

This script deploys an application to multiple servers using the -target option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and each server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppWithTargetOption(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppWithTargetOption("myApp", "/ears/DefaultApplication.ear", "myCell",
 "myNode", "myServer1", "myServer2")
```

installAppWithDeployEjbOptions

This script deploys an application with the -deployejb option.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the target node.
serverName	Specifies the name of the target server.

Syntax

```
AdminApplication.installAppWithDeployEjbOptions(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppWithDeployEjbOptions("myApp", "/ears/DefultApplication.ear", "myNode",
 "myServer")
```

installAppWithVariousTasksAndNonTasksOptions

This script deploys an application with various tasks and non-tasks options.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.

Syntax

```
AdminApplication.installAppWithVariousTasksAndNonTasksOptions(appName,
 earFile)
```

Example usage

```
AdminApplication.installAppWithVariousTasksAndNonTasksOptions("myApp",
 "/ears/DefaultApplication.ear")
```

installWarFile

This script installs a Web application archive (WAR) file. A Web module is created by assembling servlets, JavaServer Pages (JSP) files, and static content such as Hypertext Markup Language (HTML) pages into a single deployable unit. Web modules are stored in Web archive (WAR) files, which are standard Java archive files.

To run the script, specify the application name, WAR file, node name, server name, and context root arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to install.
warFile	Specifies the WAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.
contextRoot	Specifies the context root of the Web application. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

Syntax

```
AdminApplication.installWarFile(appName, warFile,
 nodeName, serverName, contextRoot)
```

Example usage

```
AdminApplication.installWarFile("myApp", "/binaries/DefaultWebApplication.war", "myNode",
 "myServer", "/")
```

uninstallApplication

This script uninstalls an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to uninstall.

Syntax

```
AdminApplication.uninstallApplication(appName)
```

Example usage

```
AdminApplication.uninstallApplication("myApp")
```

Application query scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that query your application configuration. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to query application configurations:

- “checkIfAppExists”
- “getAppDeployedNodes” on page 135
- “getAppDeploymentTarget” on page 135
- “getTaskInfoForAnApp” on page 135
- “listApplications” on page 136
- “listApplicationsWithTarget” on page 136
- “listModulesInAnApp” on page 136

checkIfAppExists

This script checks if the application is deployed on the application server.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.checkIfAppExists(appName)
```

Example usage

```
AdminApplication.checkIfAppExists("myApp")
```

getAppDeployedNodes

This script lists the nodes on which the application of interest is deployed.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.getAppDeployedNodes(appName)
```

Example usage

```
AdminApplication.getAppDeployedNodes("myApp")
```

getAppDeploymentTarget

This script displays the application deployment target for the application of interest.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.getAppDeploymentTarget(appName)
```

Example usage

```
AdminApplication.getAppDeploymentTarget("myApp")
```

getTaskInfoForAnApp

This script displays task information for a specific application Enterprise Archive (EAR) file. The script obtains information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require an update.

To run the script, specify the EAR file and the task arguments, as defined in the following table:

Argument	Description
earFile	Specifies the name of the EAR file of interest.
taskName	Specifies the name of the task of interest.

Syntax

```
AdminApplication.getTaskInfoForAnApp(appName, taskName)
```

Example usage

```
AdminApplication.getTaskInfoForAnApp("/ears/DefaultApplication.ear", "MapWebModToVH")
```

listApplications

This script lists all deployed applications. The script does not require arguments.

Syntax

```
AdminApplication.listApplications()
```

Example usage

```
AdminApplication.listApplications()
```

listApplicationsWithTarget

This script lists all deployed applications for a specific target.

To run the script, specify the node name and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminApplication.listApplicationsWithTarget(nodeName, serverName)
```

Example usage

```
AdminApplication.listApplicationsWithTarget("myNode", "server1")
```

listModulesInAnApp

This script lists each module in a deployed application.

To run the script, specify the application name and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminApplication.listModulesInAnApp(appName, serverName)
```

Example usage

```
AdminApplication.listModulesInAnApp("myApp", "myServer")
```

Application update scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that update applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to update application configurations:

- “addSingleFileToAnAppWithUpdateCommand” on page 137

addSingleFileToAnAppWithUpdateCommand

This script uses the **update** command to add a single file to a deployed application.

To run the script, specify the application name, file name, and the content uniform resource identifier (URI) arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.addSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.addSingleFileToAnAppWithUpdateCommand("myApp", "/sample.txt", "META-INFO/sample.txt")
```

addSingleModuleFileToAnAppWithUpdateCommand

This script uses the **update** command to add a single module file to a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.addSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.addSingleModuleFileToAnAppWithUpdateCommand("myApp", "/Increment.jar", "Increment.jar")
```

addUpdateSingleModuleFileToAnAppWithUpdateCommand

This script uses the **update** command to add and update a single module file for a deployed application.

To run the script, specify the application name, file name, content URI, and context root arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.
contextRoot	Specifies the context root for Web modules in the application.

Syntax

```
AdminApplication.addUpdateSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI, contextRoot)
```

Example usage

```
AdminApplication.addUpdateSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/DefaultWebApplication.war", "DefaultWebApplication.war",
"/webapp/defaultapp")
```

addPartialAppToAnAppWithUpdateCommand

This script uses the **update** command to add a partial application to a deployed application.

To run the script, specify the application name and file content arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.addPartialAppToAnAppWithUpdateCommand(appName, fileContent)
```

Example usage

```
AdminApplication.addPartialAppToAnAppWithUpdateCommand("myApp", "/partialadd.zip")
```

deleteSingleFileToAnAppWithUpdateCommand

This script uses the **update** command to delete a single file from a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deleteSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deleteSingleFileToAnAppWithUpdateCommand("myApp", "/sample.
txt", "META-INFO/sample.txt")
```

deleteSingleModuleFileToAnAppWithUpdateCommand

This script uses the **update** command to delete a single module file from a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deleteSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deleteSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/Increment.jar", "Increment.jar")
```

deletePartialAppToAnAppWithUpdateCommand

This script uses the **update** command to delete a partial application from a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deletePartialAppToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deletePartialAppToAnAppWithUpdateCommand("myApp", "/part
ialdelete.zip", "partialdelete")
```

updateApplicationUsingDefaultMerge

This script updates an application using default merging.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationUsingDefaultMerge(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationUsingDefaultMerge("myApp", "/ears/D
efaultApplication.ear")
```

updateApplicationWithUpdateIgnoreNewOption

This script updates an application using -update.ignore.new option. The system ignores the bindings from the new version of the application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationWithUpdateIgnoreNewOption(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationWithUpdateIgnoreNewOption("myApp",
"c:/ears/DefaultApplication.ear")
```

updateApplicationWithUpdateIgnoreOldOption

This script updates an application using the -update.ignore.old option. The system ignores the bindings from the installed version of the application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationWithUpdateIgnoreOldOption(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationWithUpdateIgnoreOldOption("myApp",
"/ears/DefaultApplication.ear")
```

updateEntireAppToAnAppWithUpdateCommand

This script uses the update command to update an entire deployed application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateEntireAppToAnAppWithUpdateCommand(appName, earFile)
```

Example usage

```
AdminApplication.updateEntireAppToAnAppWithUpdateCommand("myApp", "/new.ear")
```

updatePartialAppToAnAppWithUpdateCommand

This script uses the **update** command to update a partial application for a deployed application.

To run the script, specify the application name, file name,and the content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updatePartialAppToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updatePartialAppToAnAppWithUpdateCommand("myApp", "/part
ialadd.zip", "partialadd")
```

updateSingleFileToAnAppWithUpdateCommand

This script uses the **update** command to update a single file on a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updateSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updateSingleFileToAnAppWithUpdateCommand("myApp", "/sample.
txt", "META-INFO/sample.txt")
```

updateSingleModuleFileToAnAppWithUpdateCommand

This script uses the **update** command to update a single module file for a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updateSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updateSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/sample.jar", "Increment.jar")
```

Application export scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that export applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to export applications:

- "exportAnAppToFile"
- "exportAllApplicationsToDir"
- "exportAnAppDDLToDir"

exportAnAppToFile

This script exports a deployed application to a specific file.

To run the script, specify the application name and export file name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.
exportFileName	Specifies the name of the file to which the system exports the application.

Syntax

```
AdminApplication.exportAnAppToFile(appName, exportFileName)
```

Example usage

```
AdminApplication.exportAnAppToFile("myApp", "exported.ear")
```

exportAllApplicationsToDir

This script exports all deployed applications to a specific directory.

To run the script, specify the application name and export file name arguments, as defined in the following table:

Argument	Description
exportDirectory	Specifies the fully qualified directory path to which the system exports each application.

Syntax

```
AdminApplication.exportAllApplicationsToDir(exportDirectory)
```

Example usage

```
AdminApplication.exportAllApplicationsToDir("/export")
```

exportAnAppDDLToDir

This script exports the data definition language (DDL) from the application to a specific directory.

To run the script, specify the application name, export directory, and options arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to export.
exportDirectory	Specifies the fully qualified directory path to which the system exports each application.
options	Optionally specifies additional export options.

Syntax

```
AdminApplication.exportAnAppDDLToDir(appName, exportFileName, options)
```

Example usage

```
AdminApplication.exportAnAppDDLToDir("myApp", "/export", "")
```

Application deployment configuration scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that deploy applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. The application deployment script procedures contain multiple arguments. If you do not want to specify an argument with the script, specify the value of the argument as an empty string, as the following syntax demonstrates: *""*.

Use the following script procedures to deploy applications:

- "configureStartingWeightForAnApplication"
- "configureClassLoaderPolicyForAnApplication" on page 144
- "configureClassLoaderLoadingModeForAnApplication" on page 144
- "configureSessionManagementForAnApplication" on page 144
- "configureApplicationLoading" on page 145
- "configureLibraryReferenceForAnApplication" on page 146
- "configureEJBModulesOfAnApplication" on page 146
- "configureWebModulesOfAnApplication" on page 146
- "configureConnectorModulesOfAnApplication" on page 147

configureStartingWeightForAnApplication

This script configures the starting weight attribute for an application.

To run the script, specify the application name and starting weight arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
startingWeight	Specifies the starting weight to set for the application of interest.

Syntax

```
AdminApplication.configureStartingWeightForAnApplication(appName,
 startingWeight)
```

Example usage

```
AdminApplication.configureStartingWeightForAnApplication("myApp",
 "10")
```

configureClassLoaderPolicyForAnApplication

This script configures the class loader policy attribute for an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
classloaderPolicy	Specifies the class loader policy for the application of interest. For each application server in the system, you can set the application class-loader policy to `SINGLE` or `MULTIPLE`. When the application class-loader policy is set to `SINGLE`, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When the application class-loader policy is set to `MULTIPLE`, then each application receives its own class loader that is used for loading the EJB modules, dependency JAR files, and shared libraries for that application.

Syntax

```
AdminApplication.configureClassLoaderPolicyForAnApplication(appName,
 classloaderPolicy)
```

Example usage

```
AdminApplication.configureClassLoaderPolicyForAnApplication("myApp",
 "SINGLE")
```

configureClassLoaderLoadingModeForAnApplication

This script configures the class loader loading mode for an application. The class-loader delegation mode, also known as the class loader order, determines whether a class loader delegates the loading of classes to the parent class loader.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
classloaderMode	Specifies the class loader mode to set for the application of interest. You can set the class loader mode to `PARENT_FIRST` or `PARENT_LAST`. The `PARENT_FIRST` class-loader mode causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This value is the default for the class-loader policy and for standard JVM class loaders. The `PARENT_LAST` class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

Syntax

```
AdminApplication.configureClassLoaderLoadingModeForAnApplication(appName,
 classloaderMode)
```

Example usage

```
AdminApplication.configureClassLoaderLoadingModeForAnApplication("myApp",
 "PARENT_LAST")
```

configureSessionManagementForAnApplication

This script configures session management for an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
enableCookie	Specifies whether to enable cookies.
enableProtocolSwitching	Specifies whether session tracking uses cookies to carry session IDs. If cookies are enabled, session tracking recognizes session IDs that arrive as cookies and tries to use cookies for sending session IDs. If cookies are not enabled, session tracking uses Uniform Resource Identifier (URL) rewriting instead of cookies (if URL rewriting is enabled).
enableURLRewriting	Specifies whether the session management facility uses rewritten URLs to carry the session IDs. If URL rewriting is enabled, the session management facility recognizes session IDs that arrive in the URL if the encodeURL method is called in the servlet.
enableSSLTracking	**Note:** This feature is deprecated in WebSphere Application Server version 7.0. You can reconfigure session tracking to use cookies or modify the application to use URL rewriting. If you do not want to specify this argument, specify the value as an empty string, as the following syntax demonstrates: `""`. Specifies that session tracking uses Secure Sockets Layer (SSL) information as a session ID. Enabling SSL tracking takes precedence over cookie-based session tracking and URL rewriting.
enableSerializedSession	Specifies whether to allow concurrent session access in a given server.
accessSessionOnTimeout	Specifies whether the servlet is started normally or aborted in the event of a timeout. If you specify `true`, the servlet is started normally. If you specify `false`, the servlet execution aborts and error logs are generated.
maxWaitTime	Specifies the maximum amount of time a servlet request waits on an HTTP session before continuing execution. This parameter is optional and expressed in seconds. The default is 5 seconds. Under normal conditions, a servlet request waiting for access to an HTTP session gets notified by the request that currently owns the given HTTP session when the request finishes.
sessionPersistMode	Specifies whether to enable session persistence mode.
allowOverflow	Specifies whether the number of sessions in memory can exceed the value specified by the Max in-memory session count property. This option is valid only in non-distributed sessions mode.
maxInMemorySessionCount	Specifies the maximum number of sessions to maintain in memory.
invalidTimeout	Specifies the amount of time, in minutes, before an invalid timeout occurs.
sessionEnable	Specifies whether to enable session.

Syntax

```
AdminApplication.configureSessionManagementForAnApplication(appName,
 enableCookie, enableProtocolSwitching, enableURLRewriting,
 enableSSLTracking, enableSerializedSession, accessSessionOnTimeout,
 maxWaitTime, sessionPersistMode, allowOverflow,
 maxInMemorySessionCount, invalidTimeout, sessionEnable)
```

Example usage

```
AdminApplication.configureSessionManagementForAnApplication("myApplication",
 "false", "false", "true", "", "true", "90",  "NONE", "true", "1500",
 "40", "true")
```

configureApplicationLoading

This script configures the application loading attribute for an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
enableTargetMapping	Specifies whether to enable target mapping during application loading.

Syntax

```
AdminApplication.configureApplicationLoading(appName,
 enableTargetMapping)
```

Example usage

```
AdminApplication.configureApplicationLoading("myApp", "true")
```

configureLibraryReferenceForAnApplication

This script configures the library reference for an application.

To run the script, specify the application name and shared library name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
libraryName	Specifies the name of the shared library to configure.

Syntax

```
AdminApplication.configureLibraryReferenceForAnApplication(appName,
 libraryName)
```

Example usage

```
AdminApplication.configureLibraryReferenceForAnApplication("myApp",
 "sharedLibaray")
```

configureEJBModulesOfAnApplication

This script configures the EJB modules of an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to configure.
startingWeight	Specifies the target weight of the EJB modules in the application of interest.
enableTargetMapping	Specifies whether to enable target mapping for EJB modules.

Syntax

```
AdminApplication.configureEJBModulesOfAnApplication(appName,
 startingWeight, enableTargetMapping)
```

Example usage

```
AdminApplication.configureEJBModulesOfAnApplication("myApp", "1500",
 "true")
```

configureWebModulesOfAnApplication

This script configures the Web modules of an application.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.
webModuleName	Specifies the name of the Web module to configure.
startingWeight	Specifies the starting weight for the Web module of interest.

Argument	Description
classloaderMode	Specifies the class loader mode to set for the application of interest. You can set the class loader mode to PARENT_FIRST or PARENT_LAST. The PARENT_FIRST class-loader mode causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This value is the default for the class-loader policy and for standard JVM class loaders. The PARENT_LAST class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

Syntax

```
AdminApplication.configureWebModulesOfAnApplication(appName,
 webModuleName, startingWeight, classloaderMode)
```

Example usage

```
AdminApplication.configureWebModulesOfAnApplication("myApp", "myWebModule",
 "250", "PARENT_FIRST")
```

configureConnectorModulesOfAnApplication

This script configures the connector modules of an application. To run the script, specify the application name, J2C connection factory, and node name arguments.

To run the script, specify the application name argument, as defined in the following table:

Argument	Description
appName	Specifies the name of the application of interest.
j2cConnFactory	Specifies the name of the Java 2 Connector (J2C) connection factory to configure.
jndiName	Specifies the name of the Java Naming and Directory Interface (JNDI) of interest.
authDataAlias	Specifies the name of the authentication data alias of interest.
connectionTimeout	Specifies the number of seconds that a connection request waits when there are no connections available in the free pool and no new connections can be created. This usually occurs because the maximum value of connections in the particular connection pool has been reached.

Syntax

```
AdminApplication.configureConnectorModulesOfAnApplication(appName,
 j2cConnFactory, jndiName, authDataAlias,
 connectionTimeout)
```

Example usage

```
AdminApplication.configureConnectorModulesOfAnApplication("myApp",
 "myConnFactory", "myDefaultSSLSettings", "150")
```

Application administration scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that start and stop applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to start and stop applications:

- “startApplicationOnSingleServer” on page 148
- “startApplicationOnAllDeployedTargets” on page 148
- “startApplicationOnCluster” on page 148
- “stopApplicationOnSingleServer ” on page 149

startApplicationOnSingleServer

This script starts an application on a single server.

To run the script, specify the application name, node name, and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to start.
nodeName	Specifies the name of the node on which the application is deployed.
serverName	Specifies the name of the application server on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnSingleServer(appName, nodeName, serverName)
```

Example usage

```
AdminApplication.startApplicationOnSingleServer("myApp", "myNode", "myServer")
```

startApplicationOnAllDeployedTargets

This script starts an application on all deployed nodes.

To run the script, specify the application name and node name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to start.
nodeName	Specifies the name of the node on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnAllDeployedTargets(appName, nodeName)
```

Example usage

```
AdminApplication.startApplicationOnAllDeployedTargets("myApp", "myNode")
```

startApplicationOnCluster

This script starts an application on a cluster.

To run the script, specify the application name and cluster name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to start.
clusterName	Specifies the name of the cluster on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnCluster(appName, clusterName)
```

Example usage

```
AdminApplication.startApplicationOnCluster("myApp", "myCluster")
```

stopApplicationOnSingleServer

This script stops an application on a single server.

To run the script, specify the application name, node name, and server name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to stop.
nodeName	Specifies the name of the node on which the application is deployed.
serverName	Specifies the name of the application server on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnSingleServer(appName, nodeName, serverName)
```

Example usage

```
AdminApplication.stopApplicationOnSingleServer("myApp", "myNode", "myServer")
```

stopApplicationOnAllDeployedTargets

This script stops an application on all deployed nodes.

To run the script, specify the application name, cell name, and node name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to stop.
nodeName	Specifies the name of the node on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnAllDeployedTargets(appName, nodeName)
```

Example usage

```
AdminApplication.stopApplicationOnAllDeployedTargets("myApp", "myNode")
```

stopApplicationOnCluster

This script stops an application on a cluster.

To run the script, specify the application name and cluster name arguments, as defined in the following table:

Argument	Description
appName	Specifies the name of the application to stop.
clusterName	Specifies the name of the cluster on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnCluster(appName, clusterName)
```

Example usage

```
AdminApplication.stopApplicationOnCluster("myApp", "myCluster")
```

Automating business-level application configurations using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage business-level applications in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The business-level application procedures in scripting library are located in the *app_server_root*/scriptLibraries/application/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminBLA.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. See the business-level application configuration scripts documentation to view argument descriptions and syntax examples.

Use this topic and the scripting library to create an empty business-level application, add assets as composition units, and start the business-level application.

1. Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Import assets to your configuration.

   Assets represent application binaries that contain business logic that runs on the target run time environment and serves client requests. An asset can contain a file, an archive of files such as a ZIP or Java archive (JAR) file, or an archive of archive files such as a Java Platform, Enterprise Edition (Java EE) EAR file. Other examples of assets include Enterprise JavaBean (EJB) JAR files, EAR files, Service Component Architecture (SCA) composite JAR files, OSGi bundles, mediation JAR files, shared library JAR files, and non-Java EE contents such as PHP applications.

   Run the importAsset script from the AdminBLA script library to import assets to the application server configuration repository, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.importAsset("asset.zip", "true", "true")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.importAsset("asset.zip", "true", "true")
```

3. Create an empty business-level application.

   Run the createEmptyBLA script from the AdminBLA script library to create a new business-level application, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")
```

4. Add the assets, as composition units, to the business-level application.

   Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-WebSphere Application Server runtime environments without backing assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

   Run the addCompUnit script from the AdminBLA script library to add asset.zip to myBLA as a composition unit, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.addCompUnit("myBLA", "asset.zip", "default", "myCompositionUnit", "cu description", "1", "server1", "specname=actplan1")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.addCompUnit("myBLA", "asset.zip", "default", "myCompositionUnit", "cu description", "1", "server1", "specname=actplan1")
```

5. Save the configuration changes.

6. Synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to synchronize each active node in your environment, as the following example demonstrates:

```
wsadmin>AdminNodeManagement.syncActiveNodes()
```

7. Start the business-level application.

   Use the startBLA script from the AdminBLA script library to start each composition unit of the business-level application on the deployment targets for which the composition units are configured, as the following example demonstrates:

```
wsadmin>AdminBLA.startBLA("myBLA")
```

Results

The business-level application is configured and started on the deployment target of interest.

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Use the business-level application configuration scripts to create custom scripts to automate your environment. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Business-level application configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to create, query, and manage your business-level applications. You can run each script individually or combine procedures to create custom automation scripts.

The AdminBLA script procedures are located in the *app_server_root*/scriptLibraries/application/V70 directory.

Use the following script procedures to configure and administer your business-level applications:

- “addCompUnit” on page 153
- “createEmptyBLA” on page 153
- “deleteAsset” on page 153
- “deleteBLA” on page 154
- “deleteCompUnit” on page 154
- “editAsset” on page 154
- “editCompUnit” on page 155
- “exportAsset” on page 155
- “importAsset” on page 155
- “startBLA” on page 156
- “stopBLA” on page 156

Use the following script procedures to query your business-level application configurations:

- “help” on page 156
- “listAssets” on page 157
- “listBLAs” on page 157
- “listCompUnits” on page 157
- “viewAsset ” on page 158
- “viewCompUnit ” on page 158

addCompUnit

This script adds assets, shared libraries, or additional business-level applications as composition units to the empty business-level application. A composition unit represents an asset in a business-level application. A configuration unit enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents.

To run the script, specify the business-level application name and the composition unit source arguments, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application to which the system adds the composition unit.
compUnitID	Specifies the name of the composition unit to add to the business-level application of interest.
deployableUnit	Optionally specifies the name of the deployable unit for the asset. A deployable unit is the smallest portion of an asset that can be individually chosen for deployment
compUnitName	Optionally specifies the name for the composition unit to add.
compUnitDescription	Optionally specifies a description for the new composition unit.
startingWeight	Optionally specifies the starting weight of the composition unit.
target	Optionally specifies the target to which the composition unit is mapped.
activationPlan	Optionally specifies the activation plan for the composition unit.

Syntax

```
AdminBLA.addCompUnit(blaName, compUnitID, deployableUnit, compUnitName,
 compUnitDescription, startingWeight, target, activationPlan)
```

Example usage

```
AdminBLA.addCompUnit("bla1", "asset1.zip", "default", "myCompositionUnit", "cu description", "1",
"server1", "specname=actplan1")
```

createEmptyBLA

This script creates a new business-level application in your environment. Create an empty business-level application and then add assets, shared libraries, or business-level applications as composition units to the empty business-level application.

To run the script, specify the business-level application name argument, as defined in the following table:

Argument	Description
blaName	Specifies the name to assign to the new business-level application.
description	Optionally specifies a description for the business-level application.

Syntax

```
AdminBLA.createEmptyBLA(blaName, description)
```

Example usage

```
AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")
```

deleteAsset

This script removes a registered asset from your configuration.

To run the script, specify the asset ID argument, as defined in the following table:

Argument	Description
assetID	Specifies the name of the asset to delete.

Syntax

```
AdminBLA.deleteAsset(assetID)
```

Example usage

```
AdminBLA.deleteAsset("asset.zip")
```

deleteBLA

This script removes a business-level application from your configuration.

To run the script, specify the business-level application name argument, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application to delete.

Syntax

```
AdminBLA.deleteBLA(blaName)
```

Example usage

```
AdminBLA.deleteBLA("myBLA")
```

deleteCompUnit

This script removes a composition unit from a specific business-level application configuration.

To run the script, specify the business-level application name and composition unit arguments, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application of interest.
compUnitID	Specifies the identifier of the composition unit to delete.

Syntax

```
AdminBLA.deleteCompUnit(blaName, compUnitID)
```

Example usage

```
AdminBLA.deleteCompUnit("myBLA", "asset.zip")
```

editAsset

This script edits the metadata of a specific registered asset.

To run the script, specify the arguments that are defined in the following table:

Argument	Description
assetID	Specifies the name of the asset to edit.
assetDescription	Optionally specifies the new description of the asset of interest.
assetDestinationURL	Optionally specifies the new destination URL for the asset of interest.
assetTypeAspects	Optionally specifies the new type aspects for the asset of interest.
assetRelationships	Optionally specifies the new asset relationship configurations.
filePermission	Optionally specifies the new file permission configuration for the asset of interest.

Argument	Description
validateAsset	Optionally specifies whether the command validates the asset.

Syntax

```
AdminBLA.editAsset(assetID, assetDescription, assetDestinationURL,
assetTypeAspects, assetRelationships, filePermission, validateAsset)
```

Example usage

```
AdminBLA.editAsset("asset1.zip", "asset for testing", "c:/installedAssets/asset1.zip",
"WebSphere:spec=sharedlib", "", ".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755", "true")
```

editCompUnit

This script edits a specific composition unit within a business-level application.

To run the script, specify the business-level application name and composition unit ID arguments, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application to which the composition unit is associated.
compUnitID	Specifies the name of the composition unit to edit.
compUnitDescription	Optionally specifies a new description for the composition unit.
startingWeight	Optionally specifies a new starting weight for the composition unit.
target	Optionally specifies a new target to which the composition unit is mapped.
activationPlan	Optionally specifies a new activation plan for the composition unit.

Syntax

```
AdminBLA.editCompUnit(blaName, compUnitID, compUnitDescription,
startingWeight, target, activationPlan)
```

Example usage

```
AdminApplication.installAppWithDeployEjbOptions("bla1", "asset1.zip","cu description", "1",
"server1", "specname=actplan1")
```

exportAsset

This script exports a registered asset to a file on your system.

To run the script, specify the asset ID and file name arguments, as defined in the following table:

Argument	Description
assetID	Specifies the identifier of the asset to export.
fileName	Specifies the fully qualified file path to which the system exports the asset.

Syntax

```
AdminBLA.exportAsset(assetID, fileName)
```

Example usage

```
AdminBLA.exportAsset("asset.zip", "/temp/a.zip")
```

importAsset

This script imports and registers an asset to a management domain in your configuration.

To run the script, specify the assetID, displayDescription, and deployableUnit arguments, as defined in the following table:

Argument	Description
assetID	Specifies the asset to import.
displayDescription	Optionally specifies whether the script displays the description of the asset.
dispDeployableUnit	Optionally specifies whether the script displays the deployable units for the asset to import.

Syntax

```
AdminBLA.importAsset(userID, displayDescription, dispDeployableUnit)
```

Example usage

```
AdminBLA.importAsset("asset.zip", "true", "true")
```

startBLA

This script starts the business-level application process in your configuration.

To run the script, specify business-level application name argument, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application to start.

Syntax

```
AdminBLA.startBLA(blaName)
```

Example usage

```
AdminBLA.startBLA("myBLA")
```

stopBLA

This script stops the business-level application process in your configuration.

To run the script, specify the business-level application name argument, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application to stop.

Syntax

```
AdminBLA.stopBLA(blaName)
```

Example usage

```
AdminBLA.stopBLA("myBLA")
```

help

This script displays the script procedures that the AdminBLA script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminBLA.help(script)
```

Example usage

```
AdminBLA.help("createEmptyBLA")
```

listAssets

This script displays the registered assets in your configuration.

To run the script, you can choose to specify the asset ID, display description, and display deployable units arguments, as defined in the following table:

Argument	Description
assetID	Optionally specifies the group ID for which to display authorization groups.
displayDescription	Optionally specifies whether the command displays a description for each asset. Specify `true` to display descriptions.
displayDeployUnits	Optionally specifies whether the command displays the deployable units that are associated with the assets. Specify `true` to display the deployable units.

Syntax

```
AdminBLA.listAssets(assetID, displayDescription, displayDeployUnits)
```

Example usage

```
AdminBLA.listAssets("asset.zip", "true", "true")
```

listBLAs

This script displays each or specific business-level applications in your configuration.

To run the script, you can choose to specify the business-level application name and the display description arguments, as defined in the following table:

Argument	Description
blaName	Optionally specifies the name of a business-level application of interest.
displayDescription	Optionally specifies whether the command displays a description for each business-level application. Specify `true` to display descriptions.

Syntax

```
AdminBLA.listBLAs(blaName, displayDescription)
```

Example usage

```
AdminBLA.listBLAs("", "true")
```

listCompUnits

This script displays composition units within a business-level application.

To run the script, specify the business-level application name argument, as defined in the following table:

Argument	Description
blaName	Specifies the name of the authorization group of interest.
displayDescription	Optionally specifies whether the command displays a description for each composition unit. Specify `true` to display descriptions.

Syntax

```
AdminBLA.listCompUnits(blaName, displayDescription)
```

Example usage

```
AdminBLA.listCompUnits("myBLA", "true")
```

viewAsset

This script displays the configuration attributes for a specific registered asset.

To run the script, specify the asset ID argument, as defined in the following table:

Argument	Description
assetID	Specifies the name of the asset of interest.

Syntax

```
AdminBLA.viewAsset(assetID)
```

Example usage

```
AdminBLA.viewAsset("asset.zip")
```

viewCompUnit

This script displays the configuration attributes for a specific composition unit within a business-level application.

To run the script, specify the business-level application and composition unit ID arguments, as defined in the following table:

Argument	Description
blaName	Specifies the name of the business-level application of interest.
compUnitID	Specifies the identifier for the composition unit of interest.

Syntax

```
AdminBLA.viewCompUnit(blaName, compUnitID)
```

Example usage

```
AdminBLA.viewCompUnit("myBLA", "asset.zip")
```

Automating data access resource configuration using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the resource management scripts to configure and manage your Java Database Connectivity (JDBC) configurations.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  ```

```
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
    "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The data access resource management procedures in the scripting library are located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, save your automation scripts to a new subdirectory in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the scripts to perform many combinations of administration functions. Use the following sample combination of procedures to configure a JDBC provider and data source.

1. Verify that all of the necessary JDBC driver files are installed on your node manager. If you opt to configure a user-defined JDBC provider, check your database documentation for information about the driver files.
2. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the `bin` directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the `bin` directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.
3. Configure a JDBC provider.

   Run the createJDBCProvider procedure from the script library and specify the required arguments. To run the script, specify the node name, server name, name to assign to the new JDBC provider, and the implementation class name. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. Custom properties for specific vendor JDBC drivers must be set on the application server data source. Consult your database documentation for information about available custom properties.

   The following example creates a JDBC provider in your configuration:

```
bin>wsadmin -lang jython -c "AdminJDBC.createJDBCProvider("myNode",
 "myServer", "myJDBCProvider", "myImplementationClass", [["description", "testing"],
 ["xa", "false"], ["providerType", "provType"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJDBC.createJDBCProvider("myNode", "myServer",
 "myJDBCProvider", "myImplementationClass",    [["description", "testing"], ["xa",
 "false"], ["providerType", "provType"]])
```

The script returns the configuration ID of the new JDBC provider.

4. Use a template to configure a data source.

   Run the createDataSourceUsingTemplate procedure from the script library and specify the required arguments. To run the script, specify the node name, server name, JDBC provider name, configuration ID of the template to use, and the name to assign to the new data source. You can optionally specify additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

   The following example uses a template to create a data source in your configuration:

```
bin>wsadmin -lang jython -c "AdminJDBC.createDataSourceUsingTemplate("myNode",
 "myServer", "myJDBCProvider",    "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)",     "myDataSource",
 [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"],
 ["description", "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext",
 "true"],    ["statementCacheSize", "5"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJDBC.createDataSourceUsingTemplate("myNode", "myServer",
 "myJDBCProvider",    "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)",    "myDataSource",
 [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"],
 ["description", "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext",
 "true"], ["statementCacheSize", "5"]])
```

The script returns the configuration ID of the new data source.

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

J2C query scripts

The scripting library provides many script procedures to manage your Java 2 Connector (J2C) configurations. This topic provides usage information for scripts that query your J2C configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each J2C management script procedure is located in the *app_server_root*/scriptLibraries/resources/J2C directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to query your J2C configurations:

- "listAdminObjectInterfaces"
- "listConnectionFactoryInterfaces"
- "listJ2CActivationSpecs"
- "listJ2CAdminObjects" on page 162
- "listJ2CConnectionFactories" on page 162
- "listJ2CResourceAdapters" on page 162
- "listMessageListenerTypes" on page 163

listAdminObjectInterfaces

This script displays a list of the administrative object interfaces for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listAdminObjectInterfaces(resourceAdapterID)
```

Example usage

```
AdminJ2C.listAdminObjectInterfaces("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

listConnectionFactoryInterfaces

This script displays a list of the connection factory interfaces for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listConnectionFactoryInterfaces(resourceAdapterID)
```

Example usage

```
AdminJ2C.listConnectionFactoryInterfaces("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

listJ2CActivationSpecs

This script displays a list of the J2C activation specifications in your J2C configuration.

To run the script, specify the J2C resource adapter and message listener type arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
messageListenerType	Specifies the message listener type.

Syntax

```
AdminJ2C.listJ2CActivationSpecs(resourceAdapterID, messageListenerType)
```

Example usage

```
AdminJ2C.listJ2CActivationSpecs("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "javax.jms.MessageListener2")
```

listJ2CAdminObjects

This script displays a list of the administrative objects in your J2C configuration.

To run the script, specify the application name and server name arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the name of the application of interest.
adminObjectInterface	Specifies the name of the administrative object interface of interest.

Syntax

```
AdminJ2C.listJ2CAdminObjects(resourceAdapterID, adminObjectInterface)
```

Example usage

```
AdminJ2C.listJ2CAdminObjects("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "fvt.adapter.message.FVTMessageProvider2")
```

listJ2CConnectionFactories

This script displays a list of the J2C connection factories in your J2C configuration.

To run the script, specify the J2C resource adapter and connection factory interface arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
connFactoryInterface	Specifies the name of the connection factory interface of interest.

Syntax

```
AdminJ2C.listJ2CConnectionFactories(resourceAdapterID, connFactoryInterface)
```

Example usage

```
AdminJ2C.listJ2CConnectionFactories("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "javax.sql.DataSource2")
```

listJ2CResourceAdapters

This script displays each J2C resource adapter in your configuration.

To run the script, you can optionally specify the J2C resource adapter argument, as defined in the following table:

Argument	Description
resourceAdapterName	Specifies the name of the resource adapter to display.

Syntax

```
AdminJ2C.listJ2CResourceAdapters(resourceAdapterName)
```

Example usage

```
AdminJ2C.listJ2CResourceAdapters()
AdminJ2C.listJ2CResourceAdapters("myResourceAdapter")
```

listMessageListenerTypes

This script displays a list of the message listener types for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listMessageListenerTypes(resourceAdapterID)
```

Example usage

```
AdminJ2C.listMessageListenerTypes("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

J2C configuration scripts

The scripting library provides many script procedures to manage your Java 2 Connector (J2C) configurations. Use the scripts in this topic to create activation specifications, administrative objects, and connection factories, and to install resource adapters. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each J2C management script procedure is located in the *app_server_root*/scriptLibraries/resources/J2C directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to configure J2C in your environment:

- “createJ2CActivationSpec”
- “createJ2CAdminObject” on page 164
- “createJ2CConnectionFactory” on page 164
- “installJ2CResourceAdapter” on page 165

createJ2CActivationSpec

This script creates a J2C activation specification in your configuration. The script returns the configuration ID of the new J2C activation specification.

To run the script, specify the resource adapter, activation specification name, message listener type, and the Java Naming and Directory Interface (JNDI) name arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
activationSpecName	Specifies the name to assign to the new activation specification.
messageListenerType	Specifies the message listener type.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional parameters in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJ2C.createJ2CActivationSpec(resourceAdapterID,
 activationSpecName, messageListenerType, jndiName,
 attributes)
```

Example usage

```
AdminJ2C.createJ2CActivationSpec("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)
)", "J2CASTest", "javax.jms.MessageListener2", "jndiAS")
```

```
AdminJ2C.createJ2CActivationSpec("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)
)", "J2CASTest", "javax.jms.MessageListener2", "jndiAS", [["destinationJndiName", "destjndi"],
 ["description", "testing"], ["authenticationAlias", "myalias"]])
```

createJ2CAdminObject

This script creates a J2C administrative object in your configuration. The script returns the configuration ID of the new J2C administrative object.

To run the script, specify the resource adapter, activation specification name, Java Naming and Directory Interface (JNDI) name, and the administrative object interface name arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
activationSpecName	Specifies the name to assign to the new activation specification.
adminObjectInterface	Specifies the name of the administrative object interface.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional parameters in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJ2C.createJ2CAdminObject(resourceAdapterID,
 activationSpecName, adminObjectInterface, jndiName,
 attributes)
```

Example usage

```
AdminJ2C.createJ2CAdminObject("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
 "J2CAOTest", "fvt.adapter.message.FVTMessageProvider2", "jndiAO", [["description",
 "testing"]])
```

createJ2CConnectionFactory

This script creates a new J2C connection factory in your configuration. The script returns the configuration ID of the new J2C connection factory.

To run the script, To run the script, specify the resource adapter, connection factory name, the connection factory interface, and the Java Naming and Directory Interface (JNDI) name arguments, as defined in the following table:

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
connFactoryName	Specifies the name to assign to the new connection factory.
connFactoryInterface	Specifies the connection factory interface.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional parameters in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJ2C.createJ2CConnectionFactory(resourceAdapterID,
 connFactoryName, connFactoryInterface, jndiName,
 attributes)
```

Example usage

```
AdminJ2C.createJ2CConnectionFactory("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_11840917675
578)", "J2CCFTest", "javax.sql.DataSource2", "jndiCF")
```

```
AdminJ2C.createJ2CConnectionFactory("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_11840917675
578)", "J2CCFTest", "javax.sql.DataSource2", "jndiCF", [["description", "testing"], ["authDataAlias",
 "myalias"]])
```

installJ2CResourceAdapter

This script installs a J2C resource adapter in your configuration. The script returns the configuration ID of the new J2C resource adapter.

To run the script, specify the node name, resource adapter archive (RAR) file, and the resource adapter name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
rarFile	Specifies the fully qualified file path for the RAR file to install.
resourceAdapterName	Specifies the name to assign to the new resource adapter.
attributes	Optionally specifies additional parameters in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJ2C.installJ2CResourceAdapter(nodeName, rarFile,
 resourceAdapterName, attributes)
```

Example usage

```
AdminJ2C.installJ2CResourceAdapter("myNode", "/temp/jca15cmd.rar", "J2CTest")
```

```
AdminJ2C.installJ2CResourceAdapter("myNode", "/temp/jca15cmd.rar", "J2CTest",
 [["rar.desc", "testing"], ["rar.threadPoolAlias", "myalias"]])
```

JDBC configuration scripts

The scripting library provides many script procedures to manage Java Database Connectivity (JDBC) configurations in your environment. This topic provides usage information for scripts that configure JDBC settings. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJDBC script procedure is located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to configure JDBC in your environment:

- "createDataSource"
- "createDataSourceUsingTemplate"
- "createJDBCProvider" on page 167
- "createJDBCProviderUsingTemplate" on page 167

createDataSource

This script creates a new data source in your configuration. The script returns the configuration ID of the new data source.

To run the script, specify the node name, server name, JDBC provider, and data source name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name of the JDBC provider of interest.
dsName	Specifies the name to assign to the new data source.
attributes	Optionally specifies additional parameters in the following format: `[[″parameter1″, ″value1″], [″parameter2″, ″value2″]]`.

Syntax

```
AdminJDBC.createDataSource(nodeName, serverName,
 jdbcProvider, dsName, attributes)
```

Example usage

```
AdminJDBC.createDataSource("myNode", "myServer", "myJDBCProvider",
 "myDataSource")
```

```
AdminJDBC.createDataSource("myNode", "myServer", "myJDBCProvider",
 "myDataSource", [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"],
 ["description", "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext",
 "true"], ["statementCacheSize", "5"]])
```

createDataSourceUsingTemplate

This script uses a template to create a new data source in your configuration. The script returns the configuration ID of the new data source.

To run the script, specify the node name, server name, JDBC provider, template ID, and data source name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name of the JDBC provider of interest.
templateID	Specifies the configuration ID of the template to use to create the data source.
dsName	Specifies the name to assign to the new data source.
attributes	Optionally specifies additional parameters in the following format: `[[″parameter1″, ″value1″], [″parameter2″, ″value2″]]`.

Syntax

```
AdminJDBC.createDataSourceUsingTemplate(nodeName,
 serverName, jdbcProvider, templateID, dsName,
 attributes)
```

Example usage

```
AdminJDBC.createDataSourceUsingTemplate("myNode", "myServer",
 "myJDBCProvider", "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)", "myDataSource")

AdminJDBC.createDataSourceUsingTemplate("myNode", "myServer", "myJDBCProvider", "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)", "myDataSource",
 [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"], ["description",
 "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext", "true"],
 ["statementCacheSize", "5"]])
```

createJDBCProvider

This script creates a new JDBC provider in your environment. The script returns the configuration ID of the new JDBC provider.

To run the script, specify the node name, server name, JDBC provider, and implementation class arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
implementationClass	Specifies the name of the implementation class to use.
attributes	Optionally specifies additional parameters in the following format: `[[″parameter1″, ″value1″], [″parameter2″, ″value2″]]`.

Syntax

```
AdminJDBC.createJDBCProvider(nodeName, serverName,
 jdbcProvider, implementationClass, attributes)
```

Example usage

```
AdminJDBC.createJDBCProvider("myNode", "myServer", "myJDBCProvider",
 "myImplementationClass")

AdminJDBC.createJDBCProvider("myNode", "myServer", "myJDBCProvider", "myImplementationClass", [["description", "testing"], ["xa", "false"], ["providerType",
 "provType"]])
```

createJDBCProviderUsingTemplate

This script uses a template to create a new JDBC provider in your environment. To run the script, specify the node name, server name, configuration ID of the template to use, name to assign to the new JDBC provider, and the implementation class name. You can optionally specify additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`. The script returns the configuration ID of the new JDBC provider.

To run the script, specify the node name, server name, template ID, JDBC provider name, and implementation class arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
templateID	Specifies the configuration ID of the template to use to create the JDBC provider.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
implementationClass	Specifies the name of the implementation class to use.
attributes	Optionally specifies additional parameters in the following format: `[[″parameter1″, ″value1″], [″parameter2″, ″value2″]]`.

Syntax

```
AdminJDBC.createJDBCProviderUsingTemplate(nodeName,
 serverName, templateID, jdbcProvider,
 implementationClass, attributes)
```

Example usage

```
AdminJDBC.createJDBCProviderUsingTemplate("myNode", "myServer", "Derby JDBC
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#JDBCProvider_1124467079638)",
 "myJDBCProvider", "myImplementationClass")

AdminJDBC.createJDBCProviderUsingTemplate("myNode", "myServer", "Derby JDBC
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#JDBCProvider_1124467079638)",
 "myJDBCProvider", "myImplementationClass", [["description", "testing"], ["xa",
 "false"], ["providerType", "provType"]])
```

JDBC query scripts

The scripting library provides many script procedures to manage Java Database Connectivity (JDBC) configurations in your environment. This topic provides usage information for scripts that retrieve configuration IDs for your JDBC configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJDBC script procedure is located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to query your JDBC configuration:

- “listDataSources”
- “listDataSourceTemplates”
- “listJDBCProviders” on page 169
- “listJDBCProviderTemplates” on page 169

listDataSources

This script displays a list of configuration IDs for the data sources in your configuration.

No input arguments are required for the script. However, you can specify a data source name to return a specific configuration id, as defined in the following table:

Argument	Description
dsName	Optionally specifies the name of the data source of interest.

Syntax

```
AdminJDBC.listDataSources(dsName)
```

Example usage

```
AdminJDBC.listDataSources()

AdminJDBC.listDataSources("myDataSource")
```

listDataSourceTemplates

This script displays a list of configuration IDs for the data source templates in your environment.

No input arguments are required for the script. However, you can specify a template name to return a specific configuration id, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJDBC.listDataSourceTemplates(templateName)
```

Example usage

```
AdminJDBC.listDataSourceTemplates()
AdminJDBC.listDataSourceTemplates("Derby JDBC Driver DataSource")
```

listJDBCProviders

This script displays a list of configuration IDs for the JDBC providers in your environment.

No input arguments are required for the script. However, you can specify a JDBC provider name to return a specific configuration id, as defined in the following table:

Argument	Description
jdbcName	Optionally specifies the name of the JDBC provider of interest.

Syntax

```
AdminJDBC.listJDBCProviders(jdbcName)
```

Example usage

```
AdminJDBC.listJDBCProviders()
AdminJDBC.listJDBCProviders("myJDBCProvider")
```

listJDBCProviderTemplates

This script displays a list of configuration IDs for the JDBC provider templates in your environment.

No input arguments are required for the script. However, you can specify a template name to return a specific configuration id, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJDBC.listJDBCProviderTemplates(templateName)
```

Example usage

```
AdminJDBC.listJDBCProviderTemplates()
AdminJDBC.listJDBCProviderTemplates("Derby JDBC Provider")
```

Automating messaging resource configurations using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the resource management scripts to configure and manage your Java Messaging Service (JMS) configurations.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The messaging resource management procedures in the scripting library are located in the *app_server_root*/scriptLibraries/resources/JMS/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your custom Jython scripts (*.py) when the wsadmin tool starts, save your automation scripts to a new subdirectory in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the scripts to perform multiple combinations of administration functions. Use the following sample combination of procedures to create a JMS provider and configure JMS resources for the JMS provider.

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the `bin` directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the `bin` directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Configure a JMS provider.

   Run the createJMSProvider procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, external initial contextual factory name, and external provider URL. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name to assign to the new JMS provider.
External initial contextual factory name	Specifies the Java class name of the initial context factory for the JMS provider.
External provider URL	Specifies the JMS provider URL for external JNDI lookups.

   The following example creates a JMS provider in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createJMSProvider("myNode", "myServer", "myJMSProvider",
   "extInitCF", "extPURL", [["description", "testing"], ["supportsASF", "true"],
   ["providerType", "jmsProvType"]])"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createJMSProvider("myNode", "myServer", "myJMSProvider", "extInitCF",
   "extPURL", [["description", "testing"], ["supportsASF", "true"], ["providerType",
   "jmsProvType"]])
```

   The script returns the configuration ID of the new JMS provider.

3. Configure a generic JMS connection factory.

   Run the createGenericJMSConnectionFactory procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, name of the new connection factory, JNDI name, and external JNDI name. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name of the JMS provider.
Connection factory name	Specifies the name to assign to the new connection factory
JNDI name	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
External JNDI name	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

   The following example creates a JMS connection factory in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer",
  "myJMSProvider", "JMSCFTest", "jmsjndi", "extjmsjndi", [["XAEnabled", "true"],
  ["authDataAlias", "myalias"], ["description", "testing"]])"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer", "myJMSProvider",
  "JMSCFTest", "jmsjndi", "extjmsjndi", [["XAEnabled", "true"], ["authDataAlias",
  "myalias"], ["description", "testing"]])
```

   The script returns the configuration ID of the new generic JMS connection factory.

4. Create a generic JMS destination.

   Run the createGenericJMSDestination procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, generic JMS destination

name, JNDI name, and external JNDI name. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name of the JMS provider.
Generic JMS destination name	Specifies the name to assign to the new generic JMS destination.
JNDI name	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
External JNDI name	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

The following example uses a template to use a template to create a generic JMS destination in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createGenericJMSDestination("myNode", "myServer", "myJMSProvider",
   "JMSDest", "destjndi", "extDestJndi", [["description", "testing"], ["category",
   "jmsDestCatagory"], ["type", "TOPIC"]]))"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createGenericJMSDestination("myNode", "myServer", "myJMSProvider", "JMSDest",
   "destjndi", "extDestJndi", [["description", "testing"], ["category", "jmsDestCatagory"],
   ["type", "TOPIC"]]))
```

The script returns the configuration ID of the new generic JMS destination.

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

JMS configuration scripts

The scripting library provides many script procedures to manage your Java Messaging Service (JMS) configurations. This topic provides usage information for scripts that query your JMS configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJMS management script procedure is located in the *app_server_root*/scriptLibraries/resources/JMS/V70 directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to configure JMS in your environment:

createGenericJMSConnectionFactory

This script creates a new generic JMS connection factory in your configuration. The script returns the configuration ID of the new generic JMS connection factory.

To run the script, specify the node, server, JMS provider name, name of the new connection factory, JNDI name, and external JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createGenericJMSConnectionFactory(nodeName,
 serverName, jmsProvider, connFactoryName, jndiName,
 extJndiName, attributes)
```

Example usage

```
AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer",
 "JMSTest", "JMSCFTest", "jmsjndi", "extjmsjndi")

AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer",
 "JMSTest", "JMSCFTest", "jmsjndi", "extjmsjndi", [["XAEnabled", "true"],
 ["authDataAlias", "myalias"], ["description", "testing"]])
```

createGenericJMSConnectionFactoryUsingTemplate

This script uses a template to create a generic JMS connection factory in your configuration. The script returns the configuration ID of the new generic JMS connection factory.

To run the script, specify the node, server, JMS provider name, template ID, connection factory name, JNDI name, and external JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 connFactoryName, jndiName, extJndiName, attributes)
```

Example usage

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Generic QueueConnectionFactory for
 Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)", "JMSCFTest",
 "jmsjndi", "extjmsjndi")
```

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Generic QueueConnectionFactory for
 Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)", "JMSCFTest",
 "jmsjndi", "extjmsjndi", [["XAEnabled", "true"], ["authDataAlias", "myalias"],
  ["description", "testing"]])
```

createGenericJMSDestination

This script creates a generic JMS destination in your configuration. The script returns the configuration ID of the new generic JMS destination.

To run the script, specify the node, server, JMS provider name, JMS destination name, JNDI name, and external JNDI name arguments, as defined in the following table,

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createGenericJMSDestination(nodeName,
 serverName, jmsProvider, genericJMSDestination,
 jndiName, extJndiName, attributes)
```

Example usage

```
AdminJMS.createGenericJMSDestination("myNode", "myServer", "JMSTest",
 "JMSDest", "destjndi", "extDestJndi")
```

```
AdminJMS.createGenericJMSDestination("myNode", "myServer", "JMSTest",
 "JMSDest", "destjndi", "extDestJndi", [["description", "testing"], ["category",
 "jmsDestCatagory"], ["type", "TOPIC"]])
```

createGenericJMSDestinationUsingTemplate

This script uses a template to create a generic JMS destination in your configuration. The script returns the configuration ID of the new generic JMS destination.

To run the script, specify the node, server, JMS provider name, template ID, generic JMS destination name, JNDI name, and external JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createGenericJMSDestinationUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 genericJMSDestination, jndiName, extJndiName,
 attributes)
```

Example usage

```
AdminJMS.createGenericJMSDestinationUsingTemplate("myNode", "myServer",
 "JMSTest",
 "Example.JMS.Generic.Win.Topic(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_2)",
 "JMSDest", "destjndi", "extDestJndi")
```

```
AdminJMS.createGenericJMSDestinationUsingTemplate("myNode", "myServer",
 "JMSTest",
 "Example.JMS.Generic.Win.Topic(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_2)",
 "JMSDest", "destjndi", "extDestJndi", [["description", "testing"], ["category",
 "jmsDestCatagory"], ["type", "TOPIC"]])
```

createJMSProvider

This script creates a JMS provider in your configuration. The script returns the configuration ID of the new JMS provider.

To run the script, specify the node, server, JMS provider name, external initial contextual factory name, and external provider URL arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.

Argument	Description
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createJMSProvider(nodeName, serverName,
 jmsProvider, extContextFactory, extProviderURL,
 attributes)
```

Example usage

```
AdminJMS.createJMSProvider("myNode", "myServer", "JMSTest1",
 "extInitCF", "extPURL")
```

```
AdminJMS.createJMSProvider("myNode", "myServer", "JMSTest1",
 "extInitCF", "extPURL", [["description", "testing"], ["supportsASF", "true"],
 ["providerType", "jmsProvType"]])
```

createJMSProviderUsingTemplate

This script uses a template to create a JMS provider in your configuration. The script returns the configuration ID of the new JMS provider.

To run the script, specify the node, server, configuration ID of the JMS provider template, name to assign to the new JMS provider, external initial context factory, and external provider URL arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
templateID	Specifies the configuration ID of the JMS provider template to use.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createJMSProviderUsingTemplate(nodeName,
 serverName, templateID, jmsProvider,
 extContextFactory, extProviderURL, attributes)
```

Example usage

```
AdminJMS.createJMSProviderUsingTemplate("myNode", "myServer", "WebSphere JMS
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#builtin_jmsprovider)",
 "JMSTest", "extInitCF", "extPURL")
```

```
AdminJMS.createJMSProviderUsingTemplate("myNode", "myServer", "WebSphere JMS
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#builtin_jmsprovider)",
 "JMSTest", "extInitCF", "extPURL", [["description","testing"], ["supportsASF",
 "true"], ["providerType", "jmsProvType"]])
```

createWASQueue

This script creates a WAS queue in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS queue.

To run the script, specify the node, server, JMS provider name, name to assign to the queue, and JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createWASQueue(nodeName, serverName,
 jmsProvider, queueName, jndiName,
 attributes)
```

Example usage

```
AdminJMS.createWASQueue("myNode", "myServer", "JMSTest",
 "WASQueueTest", "queuejndi")

AdminJMS.createWASQueue("myNode", "myServer", "JMSTest", "WASQueueTest", "queuejndi", [["description", "testing"], ["persistence",
 "APPLICATION_DEFINED"], ["priority", "APPLICATION_DEFINED"], ["specifiedPriority",
 "2"]])
```

createWASQueueUsingTemplate

This script uses a template to WAS queue in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS queue.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the queue, and JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WAS queue template to use.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createWASQueueUsingTemplate(nodeName,
 serverName, jmsProvider, templateID, queueName,
 jndiName, attributes)
```

Example usage

```
AdminJMS.createWASQueueUsingTemplate("myNode", "myServer", "JMSTest",
 "WASQueueTest", "queuejndi")

AdminJMS.createWASQueueUsingTemplate("myNode", "myServer", "JMSTest",
 "WASQueueTest", "queuejndi", [["description", "testing"], ["persistence",
 "APPLICATION_DEFINED"], ["priority", "APPLICATION_DEFINED"], ["specifiedPriority",
 "2"]])
```

createWASQueueConnectionFactory

This script creates a WAS queue connection factory in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS queue connection factory.

To run the script, specify the node, server, JMS provider name, name to assign to the queue connection factory, and JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
queueConnFactoryName	Specifies the name to assign to the new WAS queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createWASQueueConnectionFactory(nodeName,
 serverName, jmsProvider, queueConnFactoryName,
 jndiName, attributes)
```

Example usage

```
AdminJMS.createWASQueueConnectionFactory("myNode", "myServer",
 "JMSTest", "queueCFTest", "queuejndi")
```

```
AdminJMS.createWASQueueConnectionFactory("myNode", "myServer",
 "JMSTest", "queueCFTest", "queuejndi", [["description", "testing"], ["persistence",
 "APPLICATION_DEFINED"], ["priority", "APPLICATION_DEFINED"], ["specifiedPriority",
 "2"]])
```

createWASQueueConnectionFactoryUsingTemplate

This script uses a template to create a WAS queue connection factory in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS queue connection factory.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the queue connection factory, and JNDI name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WAS queue connection factory template to use.
queueConnFactoryName	Specifies the name to assign to the new WAS queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 queueConnFactoryName, jndiName, attributes)
```

Example usage

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Example WAS
 QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
 "queueCFTest", "queuecfjndi")

AdminJMS.createWASQueueConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Example WAS
 QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
 "queueCFTest", "queuecfjndi", [['XAEnabled', 'true'], ['authDataAlias', 'myalias'],
 ['description', 'testing'], ['xaRecoveryAuthAlias', 'recoveryalias']])
```

createWASTopic

This script creates a WAS topic in your JMS configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS topic.

To run the script, specify the node, server, JMS provider name, name to assign to the topic, JNDI name, and the topic arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createWASTopic(nodeName, serverName,
 jmsProvider, topicName, jndiName, topic,
 attributes)
```

Example usage

```
AdminJMS.createWASTopic("myNode", "myServer", "JMSTest",
 "TopicTest", "topicjndi", "mytopic")

AdminJMS.createWASTopic("myNode", "myServer", "JMSTest",
 "TopicTest", "topicjndi", "mytopic", [["persistence", "PERSISTENT"], ["priority",
 "SPECIFIED"], ["description", "testing"], ["specifiedPriority", "1"]])
```

createWASTopicUsingTemplate

This script uses a template to create a WAS topic in your JMS configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS topic.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the topic, JNDI name, and the topic arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WAS topic template to use.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createWASTopicUsingTemplate(nodeName,
 serverName, jmsProvider, templateID, topicName,
 jndiName, topic, attributes)
```

Example usage

```
AdminJMS.createWASTopicUsingTemplate("myNode", "myServer", "JMSTest",
 "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)", "TopicTest",
 "topicjndi", "mytopic")

AdminJMS.createWASTopicUsingTemplate("myNode", "myServer", "JMSTest",
 "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)", "TopicTest",
 "topicjndi", "mytopic", [["persistence", "PERSISTENT"], ["priority", "SPECIFIED"],
 ["description", "testing"], ["specifiedPriority", "1"]])
```

createWASTopicConnectionFactory

This script creates a WAS topic connection factory in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS topic connection factory.

To run the script, specify the node, server, JMS provider name, name to assign to the connection factory, JNDI name, and the port arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProviderName	Specifies the name of the JMS provider of interest.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specify the port of interest.
attributes	Optionally specifies additional attributes in the following format: `[[″attr1″, ″value1″], [″attr2″, ″value2″]]`.

Syntax

```
AdminJMS.createWASTopicConnectionFactory(nodeName,
 serverName, jmsProviderName, topicConnFactoryName,
 jndiName, port, attributes)
```

Example usage

```
AdminJMS.createWASTopicConnectionFactory("myNode", "myServer",
 "JMSTest", "TopicCFTest", "topiccfjndi", "DIRECT")

AdminJMS.createWASTopicConnectionFactory("myNode", "myServer",
 "JMSTest", "TopicCFTest", "topiccfjndi", "DIRECT", [['XAEnabled', 'true'],
 ['authDataAlias', 'myalias'], ['authMechanismPreference', 'BASIC_PASSWORD'], ['clientID',
 'myID'], ['description', 'testing'], ['cloneSupport', 'true']])
```

createWASTopicConnectionFactoryUsingTemplate

This script uses a template to create a WAS topic connection factory in your configuration. You should only use WAS JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the new WAS topic connection factory.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the connection factory, JNDI name, and the port arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Argument	Description
jmsProviderName	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WAS topic connection factory to use.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specifies the port of interest.
attributes	Optionally specifies additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`.

Syntax

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProviderName, templateID,
 topicConnFactoryName, jndiName,  port, attributes)
```

Example usage

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "First Example WAS
 TopicConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
 "TopicCFTest", "topiccfjndi", "DIRECT")
```

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "First Example WAS
 TopicConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
 "TopicCFTest", "topiccfjndi", "DIRECT", [['XAEnabled', 'true'], ['authDataAlias',
 'myalias'], ['authMechanismPreference', 'BASIC_PASSWORD'], ['clientID', 'myID'],
 ['description', 'testing'], ['cloneSupport', 'true']])
```

startListenerPort

This script starts a listener port in your environment. The script returns a value of 1 if the system successfully starts the listener port or a value of -1 if the system does not start the listener port.

To run the script, specify the node and server name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminJMS.startListenerPort(nodeName,
 serverName)
```

Example usage

```
AdminJMS.startListenerPort("myNode", "myServer")
```

JMS query scripts

The scripting library provides many script procedures to manage your Java Messaging Service (JMS) configurations. This topic provides usage information for scripts that retrieve configuration IDs from your JMS configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each JMS management script procedure is located in the *app_server_root*/scriptLibraries/resources/JMS/V70 directory.

The Jython script library provides script functions for J2C resources, JDBC providers, and JMS resources at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/scriptLibraries directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Use the following script procedures to query your JMS configurations:

listGenericJMSConnectionFactories

This script displays a list of configuration IDs for the generic JMS connection factories configured in your environment.

The script does not require any input parameters. However, to return a specific generic JMS connection factory, specify the generic JMS connection factory name, as defined in the following table:

Argument	Description
connFactoryName	Optionally specifies the name of the generic JMS connection factory of interest.

Syntax

```
AdminJMS.listGenericJMSConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listGenericJMSConnectionFactories()
AdminJMS.listGenericJMSConnectionFactories("JMSCFTest")
```

listGenericJMSConnectionFactoryTemplates

This script displays a list of generic JMS connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific generic JMS connection factory template, specify the template ID argument, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listGenericJMSConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listGenericJMSConnectionFactoryTemplates()
AdminJMS.listGenericJMSConnectionFactoryTemplates("Generic QueueConnectionFactory for Windows")
```

listGenericJMSDestinations

This script displays a list of configuration IDs for the generic JMS destinations configured in your environment. The script does not require any input parameters. However, to return a specific generic JMS destination, specify the generic JMS destination name.

The script does not require any input parameters. However, to return a specific generic JMS destination, specify the generic JMS destination name, as defined in the following table:

Argument	Description
destinationName	Optionally specifies the name of the generic JMS destination of interest.

Syntax

```
AdminJMS.listGenericJMSDestinations(destinationName)
```

Example usage

```
AdminJMS.listGenericJMSDestinations()
AdminJMS.listGenericJMSDestinations("JMSDestination")
```

listGenericJMSDestinationTemplates

This script displays a list of generic JMS destination template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listGenericJMSDestinationTemplates(templateName)
```

Example usage

```
AdminJMS.listGenericJMSDestinationTemplates()
AdminJMS.listGenericJMSDestinationTemplates("Example.JMS.Generic.Win.Topic")
```

listJMSProviders

This script displays a list of configuration IDs for the JMS providers that are configured in your environment.

The script does not require any input parameters. However, to return a specific JMS provider, specify the JMS provider name, as defined in the following table:

Argument	Description
jmsProviderName	Optionally specifies the name of the generic JMS connection factory of interest.

Syntax

```
AdminJMS.listJMSProviders(jmsProviderName)
```

Example usage

```
AdminJMS.listJMSProviders()
```

```
AdminJMS.listJMSProviders("JMSTest")
```

listJMSProviderTemplates

This script displays a list of JMS provider template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listJMSProviderTemplates(templateName)
```

Example usage

```
AdminJMS.listJMSProviderTemplates()
AdminJMS.listJMSProviderTemplates("WebSphere JMS Provider")
```

listWASQueueConnectionFactoryTemplates

This script displays a list of JMS queue connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASQueueConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listWASQueueConnectionFactoryTemplates()
AdminJMS.listWASQueueConnectionFactoryTemplates("Example WAS QueueConnectionFactory")
```

listWASQueueTemplates

This script displays a list of JMS queue template configuration ids.

The script does not require any input parameters. However, to return a specific generic template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASQueueTemplates(templateName)
```

Example usage

```
AdminJMS.listWASQueueTemplates()
AdminJMS.listWASQueueTemplates("Example.JMS.WAS.Q1")
```

listWASTopicConnectionFactoryTemplates

This script displays a list of JMS topic connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASTopicConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listWASTopicConnectionFactoryTemplates()
AdminJMS.listWASTopicConnectionFactoryTemplates("First Example WAS TopicConnectionFactory")
```

listWASQueueConnectionFactories

This script displays a list of configuration IDs for the JMS queue connection factories configured in your environment.

The script does not require any input parameters. However, to return a specific JMS queue connection factory, specify the connection factory name, as defined in the following table:

Argument	Description
connFactoryName	Optionally specifies the name of the JMS connection factory of interest.

Syntax

```
AdminJMS.listWASQueueConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listWASQueueConnectionFactories()
AdminJMS.listWASQueueConnectionFactories("queuecf")
```

listWASQueues

This script displays a list of JMS queues.

The script does not require any input parameters. However, to return a specific queue, specify the queue name, as defined in the following table:

Argument	Description
queueName	Optionally specifies the name of the queue of interest.

Syntax

```
AdminJMS.listWASQueues(queueName)
```

Example usage

```
AdminJMS.listWASQueues()
AdminJMS.listWASQueues("WASQueueTest")
```

listWASTopicConnectionFactories

This script displays a list of configuration IDs for the JMS topic connection factories configured in your environment.

The script does not require any input parameters. However, to return a specific JMS topic connection factory, specify the connection factory name, as defined in the following table:

Argument	Description
connFactoryName	Optionally specifies the name of the JMS topic connection factory of interest.

Syntax

```
AdminJMS.listWASTopicConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listWASTopicConnectionFactories()
AdminJMS.listWASTopicConnectionFactories("TopicCFTest")
```

listWASTopics

This script displays a list of configuration IDs for the JMS topics configured in your environment.

The script does not require any input parameters. However, to return a specific topic, specify the topic name, as defined in the following table:

Argument	Description
topicName	Optionally specifies the name of the topic of interest.

Syntax

```
AdminJMS.listWASTopics(topicName)
```

Example usage

```
AdminJMS.listWASTopics()
AdminJMS.listWASTopics("TopicTest")
```

listWASTopicTemplates

This script displays a list of JMS topic template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASTopicTemplates(templateName)
```

Example usage

```
AdminJMS.listWASTopicTemplates()
AdminJMS.listWASTopicTemplates("Example.JMS.WAS.T1")
```

Automating authorization group configurations using the scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the authorization groups scripts create, configure, remove and query your authorization group configuration.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The authorization group management procedures in scripting library are located in the *app_server_root*/scriptLibraries/security/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminAuthorizations.py scripts to perform multiple combinations of authorization group administration functions. This topic provides one sample combination of procedures. Use the following

steps to create an authorization group, adds resources to the group, and assigns user roles.

1. Optional: Launch the wsadmin scripting tool using the Jython scripting language.

   Use this step to launch the wsadmin tool and connect to a server. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts. Alternatively, you can run each script individually without launching the wsadmin tool.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   When the wsadmin tool launches, the system loads each script from the scripting library.

2. Create an authorization group.

   Use the createAuthorizationGroup script to create a new authorization group in your configuration, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.createAuthorizationGroup("myAuthGroup")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.createAuthorizationGroup("myAuthGroup")
```

3. Add resources to the new authorization group.

   Use the addResourceToAuthorizationGroup script to add resources. You can create a file-grained administrative authorization groups by selecting administrative resources to be part of the authorization group, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

4. Assign users to the administrative role for the authorization group.

   Use the mapUsersToAdminRole script to assign one or more users to the administrative role for the resources in the authorization group. You can assign users for the authorization group to the administrator, configurator, deployer, operator, monitor, adminsecuritymanager, and iscadmins administrative roles. The following example maps the `user01`, `user02`, and `user03` users as administrators for the resources in the authorization group:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Authorization group configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to create, configure, remove and query your security authorization group configuration. You can run each script individually or combine procedures to create custom automation scripts.

The AdminAuthorizations script procedures are located in the *app_server_root*/scriptLibraries/security/V70 directory.

Use the following script procedures to configure authorization groups:

- "addResourceToAuthorizationGroup"
- "createAuthorizationGroup" on page 190
- "mapGroupsToAdminRole" on page 190
- "mapUsersToAdminRole" on page 190

Use the following script procedures to remove users and groups from the security authorization settings:

- "deleteAuthorizationGroup" on page 191
- "removeGroupFromAllAdminRoles" on page 191
- "removeGroupsFromAdminRole" on page 191
- "removeResourceFromAuthorizationGroup" on page 192
- "removeUserFromAllAdminRoles" on page 192
- "removeUsersFromAdminRole" on page 192

Use the following script procedures to query your security authorization group configuration:

- "help" on page 193
- "listAuthorizationGroups" on page 193
- "listAuthorizationGroupsForUserID" on page 193
- "listAuthorizationGroupsForGroupID" on page 193
- "listAuthorizationGroupsOfResource" on page 194
- "listUserIDsOfAuthorizationGroup" on page 194
- "listGroupIDsOfAuthorizationGroup " on page 194
- "listResourcesOfAuthorizationGroup " on page 194
- "listResourcesForUserID " on page 195

addResourceToAuthorizationGroup

This script adds a resource to an existing authorization group in your configuration. You can create a fine-grained administrative authorization groups by selecting administrative resources to be part of the authorization group. You can assign users or groups to this new administrative authorization group and also give them access to the administrative resources contained within.

To run the script, specify the authorization group name and resource name, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
resource	Specifies the name of the resource to add to the authorization group of interest.

Syntax

```
AdminAuthorizations.addResourceToAuthorizationGroup(authGroupName, resource)
```

Example usage

```
AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

createAuthorizationGroup

This script creates a new authorization group in your configuration. Administrative authorization groups that specify users and groups that have certain authorities with the selected resources.

To run the script, specify the authorization group name argument, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group to create.

Syntax

```
AdminAuthorizations.createAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.createAuthorizationGroup("myAuthGroup")
```

mapGroupsToAdminRole

This script maps group IDs to one or more administrative roles in the authorization group. The name of the authorization group that you provide determines the authorization table. The group ID can be a short name or fully qualified domain name in case Lightweight Directory Access Protocol (LDAP) user registry is used.

To run the script, specify the authorization group name, administrative role, and group ID arguments, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role to which the system maps the user IDs.
groupIDs	Specifies the group IDs to map to the role and authorization group.

Syntax

```
AdminAuthorizations.mapGroupsToAdminRole(authGroupName, adminRole, groupIDs)
```

Example usage

```
AdminAuthorizations.mapGroupsToAdminRole("myAuthGroup", "administrator", "group01 group02 group03")
```

mapUsersToAdminRole

This script maps user IDs to one or more administrative roles in the authorization group. The name of the authorization group that you provide determines the authorization table. The user ID can be a short name or fully qualified domain name in case LDAP user registry is used.

To run the script, specify the authorization group name, administrative role, and user ID arguments, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role to which the system maps the user IDs.
userIDs	Specifies the user IDs to map to the role and authorization group.

Syntax

```
AdminAuthorizations.mapUsersToAdminRole(authGroupName, adminRole, userIDs)
```

Example usage

```
AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

deleteAuthorizationGroup

This script removes an authorization group from your security configuration.

To run the script, specify the authorization group argument, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group to delete.

Syntax

```
AdminAuthorizations.deleteAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.deleteAuthorizationGroup("myAuthGroup")
```

removeGroupFromAllAdminRoles

This script removes a specific group from an administrative role in each authorization group in your configuration.

To run the script, specify the group ID argument, as defined in the following table:

Argument	Description
groupID	Specifies the group ID to remove from the administrative role in each authorization group in your configuration.

Syntax

```
AdminAuthorizations.removeGroupFromAllAdminRoles(groupID)
```

Example usage

```
AdminAuthorizations.removeGroupFromAllAdminRoles("group01")
```

removeGroupsFromAdminRole

This script removes specific groups from an administrative role in the authorization group of interest.

To run the script, specify the authorization group name, administrative role, and group ID arguments, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role from which to remove the user IDs.
groupIDs	Specifies the group IDs to remove from the specific role in the authorization group.

Syntax

```
AdminAuthorizations.removeUsersFromAdminRole(authGroupName, adminRole, groupIDs)
```

Example usage

```
AdminAuthorizations.removeUsersFromAdminRole("myAuthGroup", "administrator", "group01 group02 group03")
```

removeResourceFromAuthorizationGroup

This script removes a specific resource from the authorization group of interest.

To run the script, specify the authorization group name and resource name arguments, as defined in the following table:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
resource	Specifies the name of the resource to remove.

Syntax

```
AdminAuthorizations.removeResourceFromAuthorizationGroup(authGroupName, resource)
```

Example usage

```
AdminAuthorizations.removeResourceFromAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

removeUserFromAllAdminRoles

This script removes a specific user from an administrative role in each authorization group in your configuration.

To run the script, specify the following arguments:

Argument	Description
userID	Specifies the user ID to remove from the administrative role in each authorization group in your configuration.

Syntax

```
AdminAuthorizations.removeUserFromAllAdminRoles(userID)
```

Example usage

```
AdminAuthorizations.removeUserFromAllAdminRoles("user01")
```

removeUsersFromAdminRole

This script removes specific users from an administrative role in the authorization group of interest.

To run the script, specify the following arguments:

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role from which to remove the user IDs.
userIDs	Specifies the user IDs to remove from the specific role in the authorization group.

Syntax

```
AdminAuthorizations.removeUsersFromAdminRole(authGroupName, adminRole, userIDs)
```

Example usage

```
AdminAuthorizations.removeUsersFromAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

help

This script displays the script procedures that the AdminClusterManagement script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminResources.help(script)
```

Example usage

```
AdminResources.help("listAuthorizationGroups")
```

listAuthorizationGroups

This script displays each authorization group in your security configuration. This script does not require arguments.

Syntax

```
AdminAuthorizations.listAuthorizationGroups()
```

Example usage

```
AdminAuthorizations.listAuthorizationGroups()
```

listAuthorizationGroupsForUserID

This script displays each authorization group to which a specific user ID has access.

To run the script, specify the user ID argument, as defined in the following table:

Argument	Description
userID	Specifies the user ID for which to display authorization groups.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsForUserID(userID)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsForUserID("user01")
```

listAuthorizationGroupsForGroupID

This script displays each authorization group to which a specific group ID has access.

To run the script, specify the group ID argument, as defined in the following table:

Argument	Description
groupID	Specifies the group ID for which to display authorization groups.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsForGroupID(groupID)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsForGroupID("group01")
```

listAuthorizationGroupsOfResource

This script displays each authorization group to which a specific resource is mapped.

To run the script, specify the resource name argument, as defined in the following table:

Argument	Description
resource	Specifies the resource of interest.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsOfResource(resource)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsOfResource("Node=myNode:Server=myServer")
```

listUserIDsOfAuthorizationGroup

This script displays the user IDs and access level that are associated with a specific authorization group.

To run the script, specify the authorization group name argument, as defined in the following table:

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listUserIDsOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listUserIDsOfAuthorizationGroup("myAuthGroup")
```

listGroupIDsOfAuthorizationGroup

This script displays the group IDs and access level that are associated with a specific authorization group.

To run the script, specify the authorization group name argument, as defined in the following table:

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listGroupIDsOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listGroupIDsOfAuthorizationGroup("myAuthGroup")
```

listResourcesOfAuthorizationGroup

This script displays the resources that are associated with a specific authorization group.

To run the script, specify the authorization group name argument, as defined in the following table:

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listResourcesOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listResourcesOfAuthorizationGroup("myAuthGroup")
```

listResourcesForUserID

This script displays the resources that a specific user ID can access.

To run the script, specify the user ID argument, as defined in the following table:

Argument	Description
userID	Specifies the user ID of interest.

Syntax

```
AdminAuthorizations.listResourcesForUserID(userID)
```

Example usage

```
AdminAuthorizations.listResourcesForUserID("user01")
```

listResourcesForGroupID

This script displays the resources that a specific group ID can access.

To run the script, specify the group ID argument, as defined in the following table:

Argument	Description
groupID	Specifies the group ID of interest.

Syntax

```
AdminAuthorizations.listResourcesForGroupID(groupID)
```

Example usage

```
AdminAuthorizations.listResourcesForGroupID("group01")
```

Automating resource configurations using the scripting library.

The scripting library provides Jython script procedures to assist in automating your environment. Use the scripts in the AdminResources script library to configure mail, URL, and resource settings.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

```
#
# My Custom Jython Script - file.py
#
AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
    "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The resource management procedures in scripting library are located in the *app_server_root*/scriptLibraries/resource/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminResources.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. See the documentation for the resource configuration scripts for additional scripts, argument descriptions, and syntax examples.

The example script in this topic configures a custom mail provider and session. A mail provider encapsulates a collection of protocol providers like SMTP, IMAP and POP3, while mail sessions authenticate users and controls users' access to messaging systems. Configure your own mail providers and sessions to customize how JavaMail is handled.

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create a mail provider.

Run the createMailProvider script from the AdminResources script library, specifying the node name, server name, and new mail provider name, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.createMailProvider(myNode, myServer, newMailProvider)"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.createMailProvider(nodeName, serverName, mailProviderName)
```

3. Define the protocol provider for the mail provider.

   You can also configure custom properties, classes, JNDI name, and other mail settings with this script. See the documentation for the resource configuration scripts for argument descriptions and syntax examples. Run the configMailProvider script from the AdminResources script library to define the protocol provider, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.configMailProvider(myNode, myServer, newMailProvider, "", "", "SOAP", "", "", "", "", "", "")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.configMailProvider(myNode, myServer, newMailProvider, "", "", "SOAP", "", "", "", "", "", "")
```

4. Create the mail session.

   Run the createMailSession script from the AdminResources script library, specifying the node name, server name, mail provider name, mail session name, and Java Naming and Directory Interface (JNDI) name arguments, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.createMailSession("myNode", "myServer", "newMailProvider", "myMailSession", "myMailSession/jndi")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.createMailSession("myNode", "myServer", "newMailProvider", "myMailSession", "myMailSession/jndi")
```

5. Save the configuration changes.
6. Synchronize the node.

   To propagate the configuration changes to the node, run the syncNode script procedure from the AdminNodeManagement script library, specifying the node of interest, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminNodeManagement.syncNode("myNode")"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminNodeManagement.syncNode("myNode")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Resource configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the mail, URL, and resource environment configuration scripts to create and configure resources in your environment. You can run each script individually or combine procedures to create custom automation scripts.

The mail, URL, and resource management script procedures are located in the *app_server_root*/scriptLibraries/resources/V70 directory.

Use the following script procedures to configure your mail settings:
- "createCompleteMailProvider"
- "createMailProvider" on page 199
- "createMailSession" on page 199
- "createProtocolProvider" on page 200

Use the following script procedures to configure your resource environment settings:
- "createCompleteResourceEnvProvider" on page 200
- "createResourceEnvEntries" on page 201
- "createResourceEnvProvider" on page 201
- "createResourceEnvProviderRef" on page 202

Use the following script procedures to configure your URL provider settings:
- "createCompleteURLProvider" on page 203
- "createURL" on page 203

Use the following script procedures to configure additional Java Enterprise Edition (JEE) resources:
- "createJAASAuthenticationAlias" on page 204
- "createLibraryRef" on page 204
- "createSharedLibrary" on page 204
- "createScheduler" on page 205
- "createWorkManager" on page 206
- "help" on page 206

createCompleteMailProvider

This script configures additional configuration attributes for your mail provider. A mail provider encapsulates a collection of protocol providers like SMTP, IMAP and POP3, while mail sessions authenticate users and controls user access to messaging systems. Configure your own mail providers and sessions to customize how JavaMail is handled.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
propName	Specifies the name of the custom property.
propValue	Specifies the value of the custom property.
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
mailSessionName	Specifies the administrative name of the JavaMail session object.
JNDIName	Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.
mailStoreHost	Specifies the server that is accessed when receiving the mail. This setting, combined with the mail store user ID and password, represents a valid mail account. For example, if the mail account is `john_william@my.company.com`, then the mail store host is `my.company.com`.

Argument	Description
mailStoreUser	Specifies the user ID for the given mail account. For example, if the mail account is `john_william@my.company.com` then the user is `john_william`.
mailStorePassword	Specifies the password for the given mail account . For example, if the mail account is `john_william@my.company.com` then enter the password for ID `john_william`.

Syntax

```
AdminResources.createCompleteMailProvider(nodeName,
 serverName, mailProviderName, propName, propValue,
 protocolName, className, mailSessionName, JNDIName,
 mailStoreHost, mailStoreUser, mailStorePassword)
```

Example usage

```
AdminResources.createCompleteMailProvider("myNode",
 "myServer", "myMailProvider", "myProp", "myPropValue", "myMailProtocol",
 "com.ibm.mail.myMailProtocol.myMailStore", "myMailSession", "myMailSession/jndi", "server1",
 "mailuser", "password")
```

createMailProvider

This script creates a mail provider in your environment. The application server includes a default mail provider called the built-in provider. If you use the default mail provider you only have to configure the mail session. To use the customized mail provider you must first create the mail provider and session.

To run the script, specify the node, server, and mail provider names, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node on which to create the mail provider.
serverName	Specifies the name of the server for which to create the mail provider.
mailProviderName	Specifies the name to assign to the new mail provider.

Syntax

```
AdminResources.createMailProvider(nodeName, serverName,
 mailProviderName)
```

Example usage

```
AdminResources.createMailProvider("myNode", "myServer",
 "myMailProvider")
```

createMailSession

This script creates a new mail session for your mail provider. Mail sessions are represented by the javax.mail.Session class. A mail session object authenticates users, and controls user access to messaging systems.

To run the script, specify the node name, server name, mail provider name, mail session name, and Java Naming and Directory Interface (JNDI) name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
mailSessionName	Specifies the administrative name of the JavaMail session object.
JNDIName	Specifies the JNDI name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.

Syntax

```
AdminResources.createMailSession(nodeName, serverName,
 mailProviderName, mailSessionName, JNDIName)
```

Example usage

```
AdminResources.createMailSession("myNode", "myServer", "myMailProvider",
 "myMailSession", "myMailSession/jndi")
```

createProtocolProvider

This script creates a protocol provider in your configuration, which provides the implementation class for a specific protocol to support communication between your JavaMail application and mail servers. The application server contains protocol providers for SMTP, IMAP and POP3. If you require custom providers for different protocols, install them in your application serving environment before configuring the providers. See the JavaMail API design specification for guidelines. After configuring your protocol providers, return to the mail provider page to find the link for configuring mail sessions.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the name of the mail provider that the application server uses in association with the protocol provider.
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
type	Specifies the type of protocol provider. Valid options are `STORE` or `TRANSPORT`.

Syntax

```
AdminResources.createProtocolProvider(nodeName,
 serverName, mailProviderName, protocolName,
 className, type)
```

Example usage

```
AdminResources.createProtocolProvider("myNode", "myServer", "myMailProvider",
   "myMailProtocol", "com.ibm.mail.myMailProtocol.myMailStore",
   "STORE")
```

createCompleteResourceEnvProvider

This script configures a resource environment provider, which encapsulate the referenceables that convert resource environment entry data into resource objects in your configuration.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the name to assign to the resource environment provider.
propName	Specifies the name of a custom property to set.
propValue	Specifies the value of the custom property.
factoryClass	Specifies the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name for the referenceable.
resourceEnvEntryName	Specifies the name of the resource environment entry.
JNDIName	Specifies the JNDI name for the resource environment entry, including any naming subcontexts. This name is used as the linkage between the platform binding information for resources defined by a module deployment descriptor and actual resources bound into JNDI by the platform.

Syntax

```
AdminResources.createCompleteResourceEnvProvider(nodeName,
 serverName, resourceEnvProviderName, propName,
 propValue, factoryClass, className,
 resourceEnvEntryName, JNDIName)
```

Example usage

```
AdminResources.createCompleteResourceEnvProvider("myNode", "myServer",
 "myResEnvProvider", "myProp", "myPropValue", "com.ibm.resource.res1", "java.lang.String",
 "myResEnvEntry", "res1/myResEnv")
```

createResourceEnvEntries

This script creates a resource environment entry in your configuration. Within an application server name space, the data contained in a resource environment entry is converted into an object that represents a physical resource. This resource is frequently called an environment resource.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider for this entry. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.
resourceEnvEntry	Specifies a name for the resource environment entry to create.
JNDIName	Specifies the string to use to look up this environment resource using JNDI. This is the string to which you bind resource environment reference deployment descriptors.

Syntax

```
AdminResources.createResourceEnvEntries(nodeName,
 serverName, resourceEnvProviderName, referenceable,
 resourceEnvEntry, JNDIName)
```

Example usage

```
AdminResources.createResourceEnvEntries("myNode", "myServer",
 "myResEnvProvider", "com.ibm.resource.res1", "myResEnvEntry", "res1/myResEnv")
```

createResourceEnvProvider

This script creates a resource environment provider in your configuration. The resource environment provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.

To run the script, specify the node name, server name, and resource environment provider name arguments, as defined in the following table:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider to create.

Syntax

```
AdminResources.createResourceEnvProvider(nodeName,
 serverName, resourceEnvProviderName)
```

Example usage

```
AdminResources.createResEnvProvider("myNode", "myServer",
 "myResEnvProvider")
```

createResourceEnvProviderRef

This script creates a resource environment provider reference in your configuration. Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look up a local JEE resource. The JEE specification does not specify a particular implementation of a resource.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider for this reference. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.
factoryClass	Specifies the class of the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name to associate with the referenceable.

Syntax

```
AdminResources.createResourceEnvProviderRef(nodeName,
 serverName, resourceEnvProviderName, factoryClass,
 className)
```

Example usage

```
AdminResources.createResourceEnvProviderRef("myNode", "myServer",
 "myResEnvProvider", "com.ibm.resource.res1", "java.lang.String")
```

configURLProvider

This script configures a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.
propName	Specifies the name of the custom property to set for the URL provider.
propValue	Specifies the value of the custom property to set for the URL provider.
URLName	Specifies the name of a Uniform Resource Locator (URL) name that points to an Internet or intranet resource. For example: http://www.ibm.com.
JNDIName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.

Syntax

```
AdminResources.configURLProvider(nodeName, serverName,
 URLProviderName, URLStreamHandlerClass, URLProtocol,
 propName, propValue, URLName, JNDIName,
 URLSpec)
```

Example usage

```
AdminResources.configURLProvider("myNode", "myServer", "myURLProvider",
 "com.ibm.resource.url1", "ftp", "myProp", "myPropValue", "myURL", "url1/myURL", "myURLSpec")
```

createCompleteURLProvider

This script creates a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.

Syntax

```
AdminResources.createCompleteURLProvider(nodeName,
 serverName, URLProviderName, URLStreamHandlerClass,
 URLProtocol)
```

Example usage

```
AdminResources.createCompleteURLProvider("myNode", "myServer",
 "myURLProvider", "com.ibm.resource.url1", "ftp")
```

createURL

This script creates a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to assign the URL to.
URLName	Specifies the name of the URL to create.
JNDIName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.

Syntax

```
AdminResources.createURL(nodeName, serverName,
 URLProviderName, URLName, JNDIName,
 URLSpec)
```

Example usage

```
AdminResources.createURL("myNode", "myServer", "myURLProvider",
 "myURL", "url1/myURL", "myURLSpec")
```

createJAASAuthenticationAlias

This script creates a Java Authentication and Authorization Service (JAAS) authentication alias. The alias identifies the authentication data entry. When configuring resource adapters or data sources, the administrator can specify which authentication data to choose using the corresponding alias.

To run the script, specify the following arguments:

Argument	Description
authAliasName	Specifies the name of the authentication alias to create.
authAliasID	A user identity of the intended security domain. For example, if a particular authentication data entry is used to open a new connection to DB2, this entry contains a DB2 user identity.
authAliasPW	Specifies the password of the user identity is encoded in the configuration repository.

Syntax

```
AdminResources.createJAASAuthenticationAlias(authAliasName,
 authAliasID, authAliasPW)
```

Example usage

```
AdminResources.createJAASAuthenticationAlias("myJAAS", "user01",
 "password")
```

createLibraryRef

This script creates a library reference, which defines how to use global libraries. The first step for making a library file available to multiple applications deployed on a server is to create a shared library for each library file that your applications need. When you create the shared libraries, set variables for the library files.

To run the script, specify the following arguments:

Argument	Description
libraryRefName	Specifies the name of the library reference to create.
applicationName	Specifies the name of the application to associate with the library reference.

Syntax

```
AdminResources.createLibraryRef(libraryRefName,
 applicationName)
```

Example usage

```
AdminResources.createLibraryRef("myLibrary", "myApplication")
```

createSharedLibrary

This script creates a shared library in your configuration. The first step for making a library file available to multiple applications deployed on a server is to create a shared library for each library file that your applications need. When you create the shared libraries, set variables for the library files.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Argument	Description
sharedLibName	Specifies the name to assign to the shared library.
sharedLibClassPath	Specifies the file path where the product searches for classes and resources of the shared library. If a path in the list is a file, the product searches the contents of that `.jar` or `.zip` file. If a path in the list is a directory, then the product searches the contents of `.jar` and `.zip` files in that directory. For performance reasons, the product searches the directory itself only if the directory contains subdirectories or files other than `.jar` or `.zip` files.

Syntax

```
AdminResources.createSharedLibrary(nodeName,
 serverName, sharedLibName, sharedLibClassPath)
```

Example usage

```
AdminResources.createSharedLibrary("myNode", "myServer", "myLibrary",
 "/myLibrary.zip")
```

createScheduler

This script creates a scheduler in your configuration. Schedulers are persistent and transactional timer services that can run business logic. Each scheduler runs tasks independently and has a programming interface accessible from JEE applications using the Java Naming and Directory Interface (JNDI). You can also manage schedulers using a Java Management Extensions (JMX) MBean. See the scheduler documentation in the Information Center for details on how to configure and use schedulers.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
schedulerName	Specifies the name by which this scheduler is known for administrative purposes.
JNDIName	Specifies the JNDI name that determines where this scheduler instance is bound in the name space. Clients can look this name up directly, although the use of resource references is recommended.
scheduleCategory	Specifies a string that can be used to classify or group this scheduler.
datasourceJNDI	Specifies the name of the data source where persistent tasks are stored. Any data source available in the name space can be used with a scheduler. Multiple schedulers can share a single data source while using different tables by specifying a table prefix.
tablePrefix	Specifies the string prefix to affix to the scheduler tables. Multiple independent schedulers can share the same database if each instance specifies a different prefix string.
pollInterval	Specifies the interval, in seconds, that a scheduler polls the database. The default value is appropriate for most applications. Each poll operation can be consuming. If the interval is extremely small and there are many scheduled tasks, polling can consume a large portion of system resources. The default value is `30`.
workMgmtJNDI	Specifies the JNDI name of the work manager, which is used to manage the number of tasks that can run concurrently with the scheduler. The work manager also can limit the amount of JEE context applied to the task.

Syntax

```
AdminResources.createScheduler(nodeName, serverName,
 schedulerName, JNDIName, scheduleCategory,
 datasourceJNDI, tablePrefix, pollInterval, workMgmtJNDI)
```

Example usage

```
AdminResources.createScheduler("myNode", "myServer", "myScheduler",
 "myScheduleJndi", "Default", "jdbc/MyDatasource", "sch1", "30",
 "myWorkManager")
```

createWorkManager

This script creates a work manager in your configuration. Work managers contain a pool of threads that are bound into Java Naming and Directory Interface.

To run the script, specify the following arguments:

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
workMgrName	Specifies the name by which this work manager is known for administrative purposes.
JNDIName	Specifies the Java Naming and Directory Interface (JNDI) name used to look up the work manager in the namespace.
workMgrCategory	Specifies a string that you can use to classify or group this work manager.
alarmThreads	Specifies the desired maximum number of threads used for alarms. The default value is `2`.
minThreads	Specifies the minimum number of threads available in this work manager.
maxThreads	Specifies the maximum number of threads available in this work manager.
threadPriority	Specifies the priority of the threads available in this work manager. Every thread has a priority. Threads with higher priority are run before threads with lower priority. For more information about how thread priorities are used, see the API documentation for the setPriority method of the java.lang.Thread class in the Java EE specification.
isGrowable	Specifies whether the number of threads in this work manager can be increased. Specify a value of `true` to indicate that the number of threads can increase.
serviceNames	Specifies a list of services to make available to this work manager.

Syntax

```
AdminResources.createWorkManager(nodeName, serverName,
 workMgrName, JNDIName, workMgrCategory,
 alarmThreads, minThreads, maxThreads, threadPriority, isGrowable, serviceNames)
```

Example usage

```
AdminResources.createWorkManager("myNode", "myServer", "myWorkManager",
 "Work Manager", "wm/myWorkManager", "Default", 5, 1, 10, 5, "true",
   "AppProfileService UserWorkArea com.ibm.ws.i18n security")
```

help

This script displays the script procedures that the AdminResources script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminResources.help(script)
```

Example usage

```
AdminResources.help("createWorkManager")
```

Displaying script library help information with the wsadmin tool

The script library provides Jython script procedures to assist in automating your environment. The script library includes help commands to list each available script library, display information for specific script libraries, and to display information for specific script procedures.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER", "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication", "..\installableApps\DefaultApplication.ear",
      "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Use the AdminLibHelp script library to display general information about each script library, specific information about a specific script library, and information about specific scripts.

- Display general script library information.

  Use the following command invocation to display general script library information with the wsadmin tool:

```
print AdminLibHelp()
```

- Display scripts in a specific script library.

  You can also use AdminLibHelp script to display each script within a specific script library. For example, the following command invocation displays each script in the AdminApplication script library:

```
print AdminLibHelp.help("AdminApplication")
```

- Display detailed script information.

  Use the help script with the script library of interest to display detailed descriptions, arguments, and usage information for a specific script. For example, the following command invocation displays detailed script information for the listApplications script in the AdminApplication script library:

```
print AdminApplication.help('listApplications')
```

Chapter 4. Administering applications using scripting

You can use administrative scripts and the wsadmin tool to install, uninstall, and manage applications.

About this task

There are two methods you can use to install, uninstall, and manage applications. You can use the commands for the AdminApp and AdminControl objects to invoke operations on your application configuration.

Alternatively, you can use the AdminApplication and BLAManagement Jython script libraries to perform specific operations to configure your enterprise and business-level applications.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

You might need to complete one or more of the following topics to administer your application configurations with the wsadmin tool.

- Install enterprise applications. Use the AdminApp object or the AdminApplication script library to install an application to the application server runtime. You can install an enterprise archive file (EAR), Web archive (WAR) file, servlet archive (SAR), or Java archive (JAR) file.
- Install business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to install business-level applications.
- Manage enterprise applications using pattern matching. Use the AdminApp object or the AdminApplication script library to implement pattern matching when installing, updating, or editing an application. Pattern matching simplifies the task of supplying required values for certain complex options by allowing you to pass in asterisk (*) to all of the required values that cannot be edited.
- Manage Integrated Solutions Console applications. Use the AdminApp object to deploy or remove portlet-based Integration Solutions Console applications.
- Uninstall enterprise applications. Use the AdminApp object or the AdminApplication script library to uninstall applications.
- Uninstall business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to uninstall business-level applications.
- Switch JavaServer Faces implementations. Use the modifyJSFImplementation command to set the Sun Reference Implementation or the Apache MyFaces project as the JSF implementation for Web applications.

Installing enterprise applications using scripting

Use the AdminApp object or the AdminApplication script library to install an application to the application server runtime. You can install an enterprise archive file (EAR), Web archive (WAR) file, servlet archive (SAR), or Java archive (JAR) file.

Before you begin

On a network deployment installation, verify that the deployment manager is running before you install an application. Use the startManager command utility to start the deployment manager.

There are two ways to complete this task. Complete the steps in this topic to use the AdminApp object to install enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations.

About this task

Use this topic to install an application from an enterprise archive file (EAR), a Web archive (WAR) file, a servlet archive (SAR), or a Java archive (JAR) file. The archive file must end in `.ear`, `.jar`, `.sar` or `.war` for the wsadmin tool to complete the installation. The wsadmin tool uses these extensions to determine the archive type. The wsadmin tool automatically wraps WAR and JAR files as an EAR file.

Note: Use the most recent product version of the wsadmin tool when installing applications to mixed-version environments to ensure that the most recent wsadmin options and commands are available.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine which options to use to install the application in your configuration.

   For example, if your configuration consists of a node, a cell, and a server, you can specify that information when you enter the **install** command. Review the list of valid options for the **install** and **installinteractive** commands in the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 1269 topic to locate the correct syntax for the **-node, -cell,** and **-server** options. For this configuration, use the following command examples:

   Using Jython:

```
AdminApp.install('location_of_ear.ear','[-node nodeName -cell cellName -server serverName]')
```

   Using Jacl:

```
$AdminApp install "location_of_ear.ear" {-node nodeName -cell cellName -server serverName}
```

   You can also obtain a list of supported options for an enterprise archive (EAR) file using the **options** command, for example:

   Using Jython:

```
print AdminApp.options()
```

   Using Jacl:

```
$AdminApp options
```

3. Choose to use the **install** or **installInteractive** command to install the application.

   You can install the application in batch mode, using the **install** command, or you can install the application in interactive mode using the **installinteractive** command. Interactive mode prompts you through a series of tasks to provide information. Both the **install** command and the **installinteractive** command support the set of options you chose to use for your installation in the previous step.
4. Install the application. For this example, only the **server** option is used with the **install** command, where the value of the **server** option is `serv2`. Customize your **install** or **installInteractive** command with on the options you chose based on your configuration.
   - Using the **install** command to install the application in batch mode:
     - For a network deployment installation only, the following command uses the EAR file and the command option information to install the application on a cluster:
       - Using Jython string:

         ```
         AdminApp.install('/home/myProfile/MyStuff/application1.ear', '[-cluster cluster1]')
         ```

       - Using Jython list:

         ```
         AdminApp.install('/home/myProfile/MyStuff/application1.ear', ['-cluster', 'cluster1'])
         ```

       - Using Jacl:

         ```
         $AdminApp install "/home/myProfile/MyStuff/application1.ear" {-cluster cluster1}
         ```

       where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects to be managed
`install`	is an AdminApp command

MyStuff/application1.ear	is the name of the application to install
`cluster`	is an installation option
`cluster1`	the value of the cluster option which will be cluster name

- Use the **installInteractive** command to install the application using interactive mode. The following command changes the application information by prompting you through a series of installation tasks:
  - Using Jython:

    ```
    AdminApp.installInteractive('/home/myProfile/MyStuff/application1.ear')
    ```

  - Using Jacl:

    ```
    $AdminApp installInteractive "/home/myProfile/MyStuff/application1.ear"
    ```

  where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects to be managed
`installInteractive`	is an AdminApp command
MyStuff/application1.ear	is the name of the application to install

5. Save the configuration changes.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

What to do next

The steps in this task return a success message if the system successfully installs the application. When installing large applications, the command might return a success message before the system extracts each binary file. You cannot start the application until the system extracts all binary files. If you installed a large application, use the **isAppReady** and **getDeployStatus** commands for the AdminApp object to verify that the system extracted the binary files before starting the application.

The **isAppReady** command returns a value of `true` if the system is ready to start the application, or a value of `false` if the system is not ready to start the application. For example, using Jython:

```
print AdminApp.isAppReady('application1')
```

Using Jacl:

```
$AdminApp isAppReady application1
```

If the system is not ready to start the application, the system might be expanding application binaries. Use the **getDeployStatus** command to display additional information about the binary file expansion status, as the following examples display:

Using Jython:

```
print AdminApp.getDeployStatus('application1')
```

Using Jacl:

```
$AdminApp getDeployStatus application1
```

Setting up business-level applications using scripting

You can create an empty business-level application, and then add assets, shared libraries, or business-level applications as composition units to the empty business-level application.

Before you begin

Before you can create a business-level application, determine the assets or other files to add to your application.

Also, verify that the target application server is configured. As part of configuring the server, determine whether your application files can run on your deployment targets.

About this task

You can use the wsadmin tool to create business-level applications in your environment. This topic demonstrates how to use the AdminTask object to import and register assets, create empty business-level applications, and add assets to the business-level application as composition units. Alternatively, you can use the scripts in the AdminBLA script library to set up and administer business-level applications.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Import assets to your configuration.

   Assets represent application binaries that contain business logic that runs on the target runtime environment and serves client requests. An asset can contain an archive of files such as a compressed (zip) or Java archive (JAR) file, or an archive of archive files such as a Java Platform, Enterprise Edition (Java EE) enterprise archive (EAR) file. Examples of assets include EAR files, shared library JAR files, and custom advisors for proxy servers.

   Use the importAsset command to import assets to the application server configuration repository. See the documentation for the BLAManagement command group for the AdminTask object for additional parameter and step options.

   For this example, the commands add three assets to the asset repository. Two of the assets are non-Java EE assets and one is an enterprise asset. The following command imports the `asset1.zip` asset to the asset repository and sets the returned configuration ID to the `asset1` variable:

```
asset1 = AdminTask.importAsset('-source \ears\asset1.zip')
```

   The following command imports the `asset2.zip` asset metadata only, sets the asset name as `testAsset.zip`, sets the deployment directory, specifies that the asset is used for testing, and sets the returned configuration ID to the `testasset` variable:

```
testasset = AdminTask.importAsset('-source \ears\asset2.zip -storageType
 METADATA –AssetOptions [[.* testAsset.zip .* "asset for testing"
 c:/installedAssets/testAsset.zip/BASE/testAsset.zip "" "" "" false]]')
```

   The following command imports the `defaultapp.ear` asset, storing all application binaries, and sets the returned configuration ID to the `J2EEAsset` variable:

```
J2EEAsset = AdminTask.importAsset('-source \ears\defaultapplication.ear
 –storageType FULL –AssetOptions [[.* defaultapp.ear .* "desc" "" "" "" "" false]]')
```

   The assets of interest are registered as named configuration artifacts in the application server configuration repository, which is referred to as the asset registry. Use the listAssets command to display a list of registered assets and verify that the settings are correct, as the following example demonstrates:

```
AdminTask.listAssets('-includeDescription true -includeDeplUnit
 true')
```

3. Create an empty business-level application.

   Use the createEmptyBLA command to create a new business-level application and set the returned configuration ID to the `myBLA` variable, as the following example demonstrates:

```
myBLA = AdminTask.createEmptyBLA('-name myBLA -description "BLA that contains
 asset1, asset2, and J2EEAsset"')
```

   The system creates the business-level application. Use the listBLAs command to display a list of each business-level application in the cell, as the following example demonstrates:

```
AdminTask.listBLAs()
```

4. Add the assets, as composition units, to the business-level application.

Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-Application Server run times without backing assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

The following command adds the `asset1.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` server:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset1 -CUOptions [[.* .*
 compositionUnit1 "composition unit that is backed by asset1" 0]] -MapTargets [[.* server1]]
 –ActivationPlanOptions [[.* specname=actplan0+specname=actplan1]]')
```

The following command adds the `testAsset.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset2 -CUOptions [[.* .*
 compositionUnit2 "composition unit that is backed by asset2" 0]] -MapTargets [[.*
 server1+testServer]] –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

The following command adds the `defaultapp.ear` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('[-blaID bla1 -cuSourceID ' + J2EEAsset + '
 -defaultBindingOptions
 defaultbinding.ejbjndi.prefix=ejb#defaultbinding.virtual.host=default_host#defaultbinding.force=yes
-AppDeploymentOptions [-appname defaultapp] -MapModulesToServers [["Default Web Application" .*
 WebSphere:cell=cellName,node=nodeName,server=server1] ["Increment EJB module" .*
 Websphere:cell=cellName,node=nodeName,server=testServer]] -CtxRootForWebMod [["Default Web Application" .*
 myctx/]]]')
```

5. Save your configuration changes.
6. Synchronize the nodes.

   Use the syncActiveNodes script in the AdminNodeManagement script library to synchronize each active node in your environment, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

7. Start the business-level application.

   Use the startBLA command to start each composition unit of the business-level application on the deployment targets for which the composition units are configured, as the following example demonstrates:

```
AdminTask.startBLA('-blaID myBLA')
```

Results

The system adds three composition units backed by assets to a new business-level application. Each of the three assets are deployed and started on the `server1` server. The `testAsset.zip` and `defaultapp.ear` assets are also deployed and started on the `testServer` server.

Uninstalling enterprise applications with the wsadmin tool

You can use the AdminApp object or the AdminApplication script library to uninstall applications.

Before you begin

There are two ways to complete this task. This topic uses the AdminApp object to uninstall enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Uninstall the application:

Specify the name of the application you want to uninstall, not the name of the Enterprise Archive (EAR) file.

- Using Jacl:

```
$AdminApp uninstall application1
```

- Using Jython:

```
AdminApp.uninstall('application1')
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object supporting application objects management
`uninstall`	is an AdminApp command
`application1`	is the name of the application to uninstall

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script from the AdminNodeManagement script library to synchronize each active node in your configuration, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

Results

Uninstalling an application removes it from the application server configuration and from each server that the application was installed on. The system deletes the application binaries (EAR file contents) from the installation directory. This occurs when the configuration is saved for single server product versions or when the configuration changes are synchronized from the deployment manager to the individual nodes for network deployment configurations.

Related tasks

Uninstalling enterprise applications
After an application no longer is needed, you can uninstall it.

Chapter 4, “Administering applications using scripting,” on page 209
You can use administrative scripts and the wsadmin tool to install, uninstall, and manage applications.

“Automating application configurations using the scripting library” on page 126
The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage applications in your environment.

Related reference

“Application installation and uninstallation scripts” on page 128
The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that install applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Deleting business-level applications using scripting

You can use the wsadmin tool to remove business-level applications from your environment. Deleting a business-level application removes the application from the product configuration repository and it deletes the application binaries from the file system of all nodes where the application files are installed.

Before you begin

There are two ways to complete this task. This topic uses the commands in the BLAManagement command group for the AdminTask object to remove business-level applications from your configuration. Alternatively, you can use the scripts in the AdminBLA script library to configure, administer, and remove business-level applications

About this task

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Verify that the business-level application is ready to be deleted.

   Before deleting a business-level application, use the deleteCompUnit command to remove each configuration unit that is associated with the business-level application. Also, verify that no other business-level applications reference the business-level application to delete.

   Use the following example to delete the composition units for the business-level application of interest:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID compositionUnit1')
```

   Repeat this step for each composition unit that is associated with the business-level application of interest.
3. Delete the business-level application.

   Use the deleteBLA command to remove a business-level application from your configuration, as the following example demonstrates:

```
AdminTask.deleteBLA('-blaID myBLA')
```

   If the system successfully deletes the business-level application, the command returns the configuration ID of the deleted business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

4. Save your configuration changes.
5. Synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to each active node, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

Pattern matching with the wsadmin tool

Use the Jython or Jacl scripting language to implement pattern matching when installing, updating, or editing an application. Pattern matching simplifies the task of supplying required values for certain complex options by allowing you to pass in asterisk (*) to all of the required values that cannot be edited.

Before you begin

There are two ways to complete this task. This topic uses the AdminApp object to install enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations with many options, including pattern matching.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

- Install each Web archive (WAR) and Java archive file to the application server.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Install each Web archive (WAR) and Java archive file to the application server, as the following examples demonstrate:
     – Using Jython:

```
AdminApp.install('DefaultApplication.ear', ['-appname', 'TEST',  '-MapModulesToServers', [['.*',
 '.*', 'WebSphere:cell=myCell,node=myNode,server=myServer']]])
```

     – Using Jacl:

```
$AdminApp install DefaultApplication.ear {-appname TEST  -MapModulesToServers
  {{.* .* WebSphere:cell=myCell,node=myNode,server=myServer}}}
```

  3. Save your configuration changes.
- Install each WAR file to the `myServer` server on the `myNode`node and each JAR file to the `yourServer` server on the `yourNode` node.
  1. Launch the wsadmin scripting tool using the Jython scripting language.

2. Install the WAR and JAR files to different application server management scopes, as the following examples demonstrate:
   - Using Jython:

```
AdminApp.install('DefaultApplication.ear', ['-appname', 'TEST', '-MapModulesToServers', [['.*',
 '.*.war,.*', 'WebSphere:cell=myCell,node=myNode,server=myServer'], ['.*', '.*.jar,.*',
 'WebSphere:cell=myCell,node=yourNode,server=yourServer']]])
```

   - Using Jacl:

```
$AdminApp install DefaultApplication.ear {-appname TEST -MapModulesToServers
  {{.* .*.war,.* WebSphere:cell=myCell,node=myNode,server=myServer}
  {.* .*.jar,.* WebSphere:cell=myCell,node=yourNode,server=yourServer}}}
```

3. Save your configuration changes.

Managing administrative console applications using scripting

Use the Jython or Jacl scripting languages to deploy or remove portlet-based administrative console applications.

Before you begin

Verify that the administrative console Enterprise Archive (EAR) file is not archived before installation.

- Deploy a portlet-based console application into the EAR file.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Deploy a portlet-based console application into the EAR file.

     To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.
     - Using Jython:

```
AdminApp.update('isclite', 'modulefile', '[-operation add -contents
/WebSphere/AppServer/systemApps/isclite.ear/upzippedWarName
-contenturi upzippedWARName -usedefaultbindings -contextroot contextroot]')
```

     - Using Jacl:

```
$AdminApp update isclite modulefile {-operation add -contents
/WebSphere/AppServer/systemApps/isclite.ear/upzippedWarName
-contenturi upzippedWARName -usedefaultbindings -contextroot contextroot}
```

  3. Save your configuration changes.
- Remove a portlet-based Web archive (WAR) file.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Remove the portlet-based WAR file, as the following examples demonstrate:
     - Using Jython:

```
AdminApp.update('isclite', 'modulefile', '[-operation delete -contenturi WarName]')
```

     - Using Jacl:

```
$AdminApp update isclite modulefile {-operation delete -contenturi WarName}
```

  3. Save your configuration changes.

Managing JavaServer Faces implementations using scripting

JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java based Web applications. The product supports JSF at a runtime level, which reduces the size of Web applications since runtime binaries no longer need to be included in your Web application. Use the wsadmin tool to set the JSF implementation as the Sun Reference 1.2 implementation or the Apache MyFaces 1.2 project.

About this task

The JSF runtime:

- Makes it easy to construct a user interface from a set of reusable user interface components.
- Simplifies migration of application data to and from the user interface.
- Helps manage user interface state across server requests.
- Provides a simple model for wiring client-generated events to server-side application code.
- Supports custom user interface components to be easily build and reused.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine whether to use JSF with your applications.

   Review specification documentation for JSF 1.2 to determine whether to use JSF with your applications. Then, determine which implementation to use. You can use the Sun Reference Implementation or the open source Apache MyFaces project. The Sun Reference Implementation is the default implementation.
3. Set the JSF implementation.

   Use the modifyJSFImplementation command for the AdminTask object to set the JSF implementation.

   - The following example sets the Sun Reference Implementation for JSF:

```
AdminTask.modifyJSFImplementation('myApplication', '[-implName "SunRI1.2"]')
```

   - The following example sets the MyFaces implementation for JSF:

```
AdminTask.modifyJSFImplementation('myApplication', '[-implName "MyFaces1.2"]')
```

4. Recompile the JavaServer Pages (JSP) if you switched implementations and use precompiled JavaServer Pages (JSP) that contain JSF.

BLAManagement command group for the AdminTask object

You can use the Jython scripting language to configure and administer business-level applications with the wsadmin tool. Use the commands and parameters in the BLAManagement group to create, edit, export, delete, and query business-level applications in your configuration.

In order to configure and administer business-level applications you must use the Configurator administrative role.

An asset represents one or more application binary files that are stored in an asset repository. Typical assets include application business logic such as enterprise archives, library files, and other resource files. Use the following commands to manage your asset configurations:

- deleteAsset
- editAsset
- exportAsset
- importAsset
- listAssets
- updateAsset
- viewAsset

A business-level application is a configuration artifact that consists of zero or more composition units or other business-level applications. Business-level applications are administrative models that define an application, and can contain enterprise archive (EAR) files, shared libraries, PHP applications, and more. Use the following commands to configure and administer business-level applications:

- createEmptyBLA
- deleteBLA
- editBLA
- getBLAStatus
- listBLAs
- listControlOps
- startBLA
- stopBLA
- viewBLA

A composition unit represents an asset in a business-level application. A configuration unit enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents. Use the following commands to manage your composition unit configurations:

- addCompUnit
- deleteCompUnit
- editCompUnit
- listCompUnits
- setCompUnitTargetAutoStart
- viewCompUnit

deleteAsset

The deleteAsset command removes an asset from your business-level application configuration. Before using this command, verify that no composition units are associated with the asset of interest. The command fails if the asset is associated with configuration units.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to delete. The command accepts incomplete IDs for the assetID parameter, as long as the system can match the string to a unique asset. (String, required)

Optional parameters

-force
Specifies whether to force the system to delete the asset, even if other assets depend on this asset. (Boolean, optional)

Return value

The command returns the configuration ID of the deleted asset, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAsset('-assetID asset2.zip -force true')
```

- Using Jython list:

```
AdminTask.deleteAsset(['-assetID', 'asset2.zip', '-force', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAsset('-interactive')
```

editAsset

The editAsset command modifies additional asset configuration options. You can use this command to modify the description, destination URL, asset relationships, file permissions, and validation settings.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to edit. This parameter accepts an incomplete configuration ID, as long as the system can match the string to a unique asset ID. (String, required)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-AssetOptions
Use the AssetOptions step and the following arguments to set additional properties for the asset.

inputAsset (read-only)
Specifies the source package of the asset.

name (read-only)
Specifies the name of the asset. The default value for this argument is the file name of the source package.

defaultBindingProps (read-only)
Specifies the default binding properties for the asset. This argument only applies to enterprise assets. For assets which are not enterprise assets, specify the asterisk character (*) for pattern matching. For enterprise assets, specify the `.*` value to set the argument as a non-empty value.

description
Specifies a description for the asset.

destinationUrl
Specifies the URL of the asset binaries to deploy.

typeAspect
Specifies the asset type aspect.

relationship
Specifies the asset relationship. Use the plus sign character (+) to add additional assets to the existing relationship. Use the number sign character (#) to delete an existing asset from the relationship. To replace the existing relationships, specify the same syntax as in the importAsset command. If the asset specified in the relationship does not exist for add or update, the command returns an exception.

filePermission
Specifies the file permission configuration.

validate
Specifies whether to validate the asset. The default value is `false`.

Return value

The command returns the configuration ID of the asset of interest.

Batch mode example usage

Use the following examples to edit a non-enterprise asset:

- Using Jython string:

```
AdminTask.editAsset('-assetID asset3.zip –AssetOptions [[.* asset3.zip * "asset for testing"
   c:/installedAssets/asset3.zip/BASE/asset3.zip "" assetname=a.jar "" false]]')
```

- Using Jython list:

```
AdminTask.editAsset(['-assetID', 'asset3.zip', '–AssetOptions', '[[.* asset3.zip * "asset for testing"
   c:/installedAssets/asset3.zip/BASE/asset3.zip "" assetname=a.jar "" false]]'])
```

Use the following examples to edit an enterprise asset:

- Using Jython string:

```
AdminTask.editAsset('-assetID defaultapp.ear –AssetOptions
  [[.* defaultapp.ear .* "asset for testing" "" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.editAsset(['-assetID', 'defaultapp.ear', '–AssetOptions',
   '[[.* defaultapp.ear .* "asset for testing" "" "" "" "" false]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editAsset('-interactive')
```

exportAsset

The exportAsset command exports an asset configuration to a file.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to export. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

-filename
Specifies the file name to which the system exports the asset configuration. (DownloadFile, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportAsset('-assetID asset2.zip –filename c:/temp/a2.zip')
```

- Using Jython list:

```
AdminTask.exportAsset(['-assetID', 'asset2.zip', '-filename', 'c:/temp/a2.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportAsset('-interactive')
```

importAsset

The importAsset command imports an asset configuration to the asset repository. After importing assets, you can add the assets to business-level applications as composition units.

Target object

None

Required parameters

-source
Specifies the name of the source file to import. (UploadFile, required)

Optional parameters

-storageType
Specifies the way the system saves the asset in the asset repository. The default asset repository stores full binaries, metadata of binaries, or no binaries. Specify `FULL` to store full binaries. Specify `METADATA` to store the metadata portion of the binaries. Specify `NONE` to store no binaries in the asset repository. The default value is `FULL`. (String, optional)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-AssetOptions
Use the AssetOptions step and the following arguments to set additional properties for the asset.

inputAsset (read-only)
Specifies the source package of the asset.

name Specifies the name of the asset. The extension file name of the asset must match the extension file name of the source package. The default value for this argument is the file name of the source package.

defaultBindingProps (read-only)
Specifies the default binding properties for the asset. This argument only applies to enterprise assets. For assets which are not enterprise assets, specify the asterisk character (*) for pattern matching. For enterprise assets, specify the `.*` value to set the argument as a non-empty value.

description
Specifies a description for the asset.

destinationUrl
Specifies the URL of the asset binaries to deploy.

typeAspect
Specifies the asset type aspect. Specify the typeAspect option in object name format, as the following syntax demonstrates: `spec=xxx`

relationship
Specifies the asset relationship. Use the plus sign character (+) to specify multiple asset relationships. The command returns an exception if you specify assets in the relationship that do not exist.

filePermission
Specifies the file permission configuration.

validate
Specifies whether to validate the asset.

Return value

The command returns the configuration ID of the asset that the system creates, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

Batch mode example usage

Use the following examples to import a non-enterprise asset:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\asset1.zip -storageType NONE')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset1.zip', '-storageType', 'NONE'])
```

Use the following examples to import a non-enterprise asset, set `asset2.zip` as the asset name, save the metadata binaries in the asset repository, and set the destination directory of the binaries to deploy:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\asset1.zip -storageType METADATA –AssetOptions
  [[.* asset2.zip .* "asset for testing" c:/installedAssets/asset2.zip/BASE/asset2.zip "" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset1.zip', '-storageType', 'METADATA', '–AssetOptions',
   '[[.* asset2.zip .* "asset for testing" c:/installedAssets/asset2.zip/BASE/asset2.zip "" "" "" "" false]]')
```

Use the following examples to import a non-enterprise asset, and specifies asset relationships with the `a.jar` and `b.jar` assets:

- Using Jython string:

```
AdminTask.importAsset('[-source c:\ears\asset3.zip -storageType FULL –AssetOptions
   [[.* asset3.zip .* "asset for testing" "" spec=zip assetname=a.jar+assetname=b.jar "" false]]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset3.zip', '-storageType', 'FULL', '–AssetOptions',
  '[[.* asset3.zip .* "asset for testing" "" spec=zip assetname=a.jar+assetname=b.jar "" false]]'])
```

Use the following examples to import an enterprise asset:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\defaultapplication.ear –storageType FULL –AssetOptions
   [[.* defaultapp.ear .* "desc" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\defaultapplication.ear', '–storageType',
  'FULL', '–AssetOptions', '[[.* defaultapp.ear .* "desc" "" "" "" false]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importAsset('-interactive')
```

listAssets

The listAssets command displays the configuration ID of each asset within the cell.

Target object

None

Optional parameters

-assetID
Specifies the configuration ID of the asset of interest. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, optional)

-includeDescription
Specifies whether to include the a description of each asset that the command returns. Specify `true` to display the asset descriptions. (String, optional)

-includeDeplUnit
Specifies whether to display the deployable units for each asset that the command returns. Specify `true` to display the deployable units. (String, optional)

Return value

The command returns a list of configuration IDs for the assets of interest. Depending on the parameter values specified, the command might display the description and deployable composition units for each asset, as the following example displays:

```
WebSphere:assetname=asset1.zip
"asset for testing"

WebSphere:assetname=asset2.zip
"second asset for testing"
a.jar

WebSphere:aasetname=asset3.zip
"third asset for testing"
a1.jar+a2.jar

WebSphere:assetname=a.jar0
"a.jar for sharedlib"

WebSphere:assetname=b.jar
"b.jar for sharedlib"

WebSphere:assetname=defaultapp.ear
"default app"
```

Batch mode example usage

Use the following examples to list each asset in the cell:

- Using Jython:

```
AdminTask.listAssets()
```

Use the following examples to list each asset in the cell:

- Using Jython string:

```
AdminTask.listAssets('-assetID asset1.zip')
```

- Using Jython list:

```
AdminTask.listAssets(['-assetID asset1.zip'])
```

Use the following examples to list each asset, asset description, and deployable composition units in the cell:

- Using Jython string:

```
AdminTask.listAssets('-includeDescription true –includeDeplUnit true')
```

- Using Jython list:

```
AdminTask.listAssets(['-includeDescription', 'true', '–includeDeplUnit', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAssets('-interactive')
```

updateAsset

The updateAsset command modifies one or more files or module files or module files of an asset. The command updates the asset binary file, but does not update the composition units that the system deploys with the asset as a backing object.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to update. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

-operation
Specifies the operation to invoke on the asset of interest. (String, required)

The following table displays each operation that you can invoke on an asset:

Operation	Description
`replace`	The `replace` operation replaces the contents of the asset of interest.
`merge`	The `merge` operation updates multiple files for the asset, but does not update all files.
`add`	The `add` operation adds a new file or module file.
`addupdate`	The `addupdate` operation adds or updates one file or module file. If the file does not exist, the system adds the contents. If the file exists, the system updates the file.
`update`	The `update` operation updates one file or module file.
`delete`	The `delete` operation deletes a file or module file.

-contents
Specifies the file that contains the content to add or update. This parameter is not required for the `delete` operation. (UploadFile, optional)

Optional parameters

-contenturi
Specifies the Uniform Resource Identifier (URI) of the file to add, update, or remove from the asset. This parameter is not required for the `merge` or `replace` operations. (String, optional)

Return value

The command returns

Batch mode example usage

The following example replaces the contents of a non-enterprise asset:

- Using Jython string:

```
AdminTask.updateAsset('-assetID asset1.zip -operation replace -contents c:/temp/a.zip')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'asset1.zip', '-operation', 'replace', '-contents', 'c:/temp/a.zip'])
```

The following example partially updates the files of a non-enterprise asset:

- Using Jython string:

```
AdminTask.updateAsset('-assetID asset1.zip –operation merge –contents c:/temp/p.zip')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'asset1.zip', '–operation', 'merge', '–contents', 'c:/temp/p.zip'])
```

The following example updates an enterprise asset with an Enterprise JavaBean (EJB) module file:

- Using Jython string:

```
AdminTask.updateAsset('-assetID defaultapp.ear –operation add –contents
  c:/temp/filename.jar –contenturi filename.jar')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'defaultapp.ear', '–operation', 'add', '–contents',
  'c:/temp/filename.jar', '–contenturi', 'filename.jar'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updateAsset('-interactive')
```

viewAsset

The viewAsset command displays additional asset configuration options and configured values.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset of interest. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

Optional parameters

None

Return value

The command returns configuration data for the asset of interest, as the following example displays:

```
Specify Asset options (AssetOptions)

Specify options for Asset.

*Asset Name (name): [defaultapp.ear]
Default Binding Properties (defaultBindingProps):
 [defaultbinding.ejbjndi.prefix#defaultbinding.datasource.jndi#defaultbinding.datasource.username
#defaultbinding.datasource.password#defaultbinding.cf.jndi
#defaultbinding.cf.resauth#defaultbinding.virtual.host#defaultbinding.force]
Asset Description (description): []
Asset Binaries Destination Url (destination): [${USER_INSTALL_ROOT}/installedAssets/defaultapp.ear/BASE/defaultapp.ear]
Asset Type Aspects (typeAspect): [WebSphere:spec=j2ee_ear]
Asset Relationships (relationship): []File Permission (filePermission): [.*\\.dll=755#.*\\.so=755#.*\\.a=755#.*\\.sl=755]
Validate asset (validate): [false]
```

Batch mode example usage

- Using Jython string:

```
AdminTask.viewAsset('-assetID asset3.zip')
```

- Using Jython list:

```
AdminTask.viewAsset(['-assetID', 'asset3.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewAsset('-interactive')
```

addCompUnit

The addCompUnit command adds a composition unit to a specific business-level application. A composition unit represents an asset in a business-level application, and enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuSourceID
Specifies the source configuration ID for the composition unit to add. You can specify an asset ID or a business-level application ID. (String, required)

Optional parameters

-deplUnits
Specifies the deployable units to deploy for the asset. You can specify a subset of deployable units, all deployable units, or use the default as a shared library. If you do not specify this parameter, the system deploys each deployable unit. The system does not deploy Java EE assets. (String, optional)

-cuConfigStrategyFile
Specifies the fully qualified file path for custom default binding properties. This parameter only applies to enterprise assets. (String, optional)

-defaultBindingOptions
Specifies optional Java Naming and Directory Interface (JNDI) binding properties for an enterprise asset. The binding properties available depend upon the type of enterprise asset. Use the format `property=value` to specify a default binding property. To specify more than one property, separate each `property=value` statement by the delimiter `#`.

You can specify binding properties now, when creating the asset, or later, when adding the asset as a composition unit to a business-level application. If you specify binding properties later, when adding the asset to a business-level application, then you can use a strategy file to specify the binding properties. (String, optional)

Use the following options with the defaultBindingOptions parameter:

enterprise asset type	Supported binding properties

Enterprise bean (EJB)	`defaultbinding.ejbjndi.prefix`
	
	`defaultbinding.force`

Data source	`defaultbinding.datasource.jndi`
	
	`defaultbinding.datasource.username`
	
	`defaultbinding.datasource.password`
	
	`defaultbinding.force`

enterprise asset type	Supported binding properties
Connection factory	defaultbinding.cf.jndi defaultbinding.cf.resauth defaultbinding.force
Virtual host	defaultbinding.virtual.host defaultbinding.force

Optional steps

You can also specify values for optional steps to set additional properties for the new composition unit. These steps do not apply to enterprise assets. For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-CUOptions
Specifies additional properties for the composition unit. Specify the following options with the CUOptions step:

parentBLA (read-only)
Specifies the parent business-level application for the new composition unit.

backingID (read-only)
Specifies the composition unit source ID.

name Specifies the name of the composition unit.

description
Specifies a description of the composition unit.

startingWeight
Specifies the starting weight of the composition unit.

startedOnDistributed
Specifies whether to start the composition unit after distributing changes to the target nodes. The default value is `false`.

restartBehaviorOnUpdate
Specifies the nodes to restart after editing the composition unit. Specify `ALL` to restart each target node. Specify `DEFAULT` to restart the nodes controlled by the sync plug-ins. Specify `NONE` to prevent the system from restarting nodes.

For example, specify the syntax of this step as `-CUOptions [[.* .* cu4 "cu4 desc" 0 false DEFAULT]]`

-MapTargets
Specifies additional properties for the composition unit target mapping. Specify the following options with the MapTargets step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

server Specifies the target or targets to deploy the composition units. The default value is the `server1` server. Use the plus sign character (+) to specify multiple targets. Use the plus sign character (+) as a prefix to add an additional target. Specify the complete object name format for each server that is not aWebSphere Application Server server.

For example, specify the syntax of this step as `-MapTargets [[a1.jar cluster1+cluster2] [a2.jar +server2]]`

-ActivationPlanOptions

Specifies additional properties for the composition unit activation plan. Specify the following options with the ActivationPlanOptions step:

deplUnit (read-only)

Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

activationPlan

Specifies a list of components as the activation plan. Specify each activation plan in the format `specName=xxx,specVersion=yyy`, where `specName` represents the name of the specification and is required. Use the plus sign character (+) to specify multiple activation plans.

For example, specify the syntax of this step as `-ActivationPlanOptions [[a1.jar specname=actplan0+specname=actplan1] [a2.jar specname=actplan1+specname=actplan2]]`

-CreateAuxCUOptions

Specifies additional properties for an auxiliary composition unit. Use this step if the composition unit source is an asset that corresponds to an asset that does not have a matching composition unit in the business-level application. Specify the following options with the CreateAuxCUOptions step:

deplUnit (read-only)

Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

inputAsset (read-only)

Specifies composition unit source ID.

cuID Specifies the composition unit ID that the system creates for the asset. If you do not want to create a new composition unit, do not specify this argument.

matchTarget

Specifies whether to match the targets of the dependency auxiliary composition unit with the targets of the new composition unit. The default value is `true`.

For example, specify the syntax of this step as `–CreateAuxCUOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

-RelationshipOptions

Specifies additional properties for relationships between assets, composition units, and business-level applications. Use this step if the source ID of the composition unit is an asset that has a matching composition unit in the business-level application. Specify the following options with the RelationshipOptions step:

deplUnit (read-only)

Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

relationship

Defines the composition unit relationships. Specify the composition unit object name in the format: `cuName=xxx`. Use the plus sign character (+) to specify multiple composition unit object names in the relationship. If the composition unit specified in the relationship does not exist under the same business-level application, the system removes the composition unit from the relationship.

matchTarget

Specifies whether to match the targets of the composition unit relationship with the targets of the new composition unit. The default value is `true`.

For example, specify the syntax of this step as `–RelationshipOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

Return value

The command returns the configuration IDs of the composition unit and the new composition unit created for the asset in the asset relationship, as the following example displays:

```
WebSphere:cuname=cu4
WebSphere:cuname=cua
WebSphere:cuname=cud
```

Batch mode example usage

Use the following examples to add a non-enterprise asset:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID assetname=asset1.zip -CUOptions
  [[.* .* cu1 "cu1 desc1" 0 false DEFAULT]] -MapTargets [[.* server1]] –ActivationPlanOptions
  [.* specname=actplan0+specname=actplan1]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'myBLA', '–cuSourceID', 'assetname=asset1.zip', '-CUOptions',
  '[[.* .* cu1 "cu1 desc1" 0 false DEFAULT]]', '-MapTargets', '[[.* server1]]', '–ActivationPlanOptions',
   '[.* specname=actplan0+specname=actplan1]'])
```

Use the following examples to add a business-level application composition unit:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID myBLA -cuSourceID yourBLA -CUOptions [[.* .* cu3 "cu3 desc3" 0 false DEFAULT]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'myBLA', '-cuSourceID', 'yourBLA', '-CUOptions',
  '[[.* .* cu3 "cu3 desc3" 0 false DEFAULT]]'])
```

Use the following examples to add a composition unit for a non-enterprise asset and deploy the composition unit to multiple targets:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID theirBLA –cuSourceID asset2.zip –CUOptions
  [[.* .* cu2 "cu2 desc" 0 false DEFAULT]] -MapTargets [[.* server1+server2]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'theirBLA', '–cuSourceID', 'asset2.zip', '–CUOptions',
  '[[.* .* cu2 "cu2 desc" 0 false DEFAULT]]', '-MapTargets', '[[.* server1+server2]]'])
```

Use the following examples to add a composition unit that is a non-enterprise asset with a deployable unit:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID yourBLA –cuSourceID asset2.zip –deplUnits a.jar –CUOptions
  [[.* .* cu3 "cu3 desc" 0 false DEFAULT]] –MapTargets [[a.jar server1]] –ActivationPlanOptions
  [[a.jar specname=actplan1]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'yourBLA', '–cuSourceID', 'asset2.zip', '–deplUnits', 'a.jar', '–CUOptions',
  '[[.* .* cu3 "cu3 desc" 0 false DEFAULT]]', '–MapTargets', '[[a.jar server1]]', '–ActivationPlanOptions',
   '[[a.jar specname=actplan1]]'])
```

Use the following examples to add a composition unit for a non-enterprise asset as a shared library:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID ourBLA –cuSourceID b.jar –deplUnits default –CUOptions
  [[.* .* cub "cub desc" 0 false DEFAULT]] –MapTargets [[default server1]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'ourBLA', '–cuSourceID', 'b.jar', '–deplUnits', 'default', '–CUOptions',
  '[[.* .* cub "cub desc" 0 false DEFAULT]]', '–MapTargets', '[[default server1]]'])
```

Use the following examples to add a composition unit for a non-enterprise asset with a dependency. For this example, the cub composition unit exists as a shared library of the ourBLA business-level application:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID ourBLA –cuSourceID asset3.zip –deplUnits a1.jar –CUOptions
  [[.* .* cu4 "cu4 desc" 0 false DEFAULT]] –MapTargets [[a1.jar cluster1+cluster2]] –CreateAuxCUOptions
  [[a1.jar a.jar cua true]] –RelationshipOptions [[a1.jar cuname=cub true]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'ourBLA', '–cuSourceID', 'asset3.zip', '–deplUnits', 'a1.jar', '–CUOptions',
  '[[.* .* cu4 "cu4 desc" 0 false DEFAULT]]', '–MapTargets', '[[a1.jar cluster1+cluster2]]', '–CreateAuxCUOptions',
  '[[a1.jar a.jar cua true]]', '–RelationshipOptions', '[[a1.jar cuname=cub true]]'])
```

Use the following examples to add an enterprise asset:

- Using Jython string:

```
AdminTask.addCompUnit('[-blaID yourBLA –cuSourceID defaultapp.ear –defaultBindingOptions
  defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host#
  defaultbinding.force=yes –AppDeploymentOptions [-appname defaultapp -installed.ear.destination
  application_root/myCell/defaultapp.ear] –MapModulesToServers
  [[defaultapp.war .* WebSphere:cell=cellName,node=nodeName,server=server1]
  [Increment.jar .* Websphere:cell=cellName,node=nodeName,server=server2]] -CtxRootForWebMod
  [[defaultapp.war .* myctx/]]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'yourBLA', '–cuSourceID', 'defaultapp.ear', '–defaultBindingOptions',
   'defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host# defaultbinding.force=yes',
   '–AppDeploymentOptions', '[-appname defaultapp -installed.ear.destination application_root/myCell/defaultapp.ear]',
   '–MapModulesToServers', '[[defaultapp.war .* WebSphere:cell=cellName,node=nodeName,server=server1]
   [Increment.jar .* Websphere:cell=cellName,node=nodeName,server=server2]]', '-CtxRootForWebMod',
   '[[defaultapp.war .* myctx/]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addCompUnit('-interactive')
```

deleteCompUnit

The deleteCompUnit command removes a composition unit. Both parameters for this command accept incomplete configuration IDs, as long as the system can match the string to a unique ID.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuID
Specifies the configuration ID of the composition unit to delete. (String, required)

Optional parameters

-force
Specifies whether to force the system to delete the composition unit, whether or not other composition units are associated with it. (Boolean, optional)

Return value

The command returns the configuration ID of the composition unit that the system deleted, as the following example displays:

```
WebSphere:cuname=cu1
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1 -force true')
```

- Using Jython list:

```
AdminTask.deleteCompUnit(['-blaID', 'myBLA', '–cuID', 'cu1', '-force', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteCompUnit('-interactive')
```

editCompUnit

The editCompUnit command modifies additional composition unit options. You can use this command to modify the starting weight of the composition unit, deployment targets, activation plan options, and relationship settings.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuID
Specifies the configuration ID of the composition unit to edit. (String, required)

Optional steps

You can also specify values for optional steps to edit properties of the composition unit. These steps do not apply to enterprise assets. For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-CUOptions
Specifies additional properties for the composition unit. Specify the following options with the CUOptions step:

parentBLA (read-only)
Specifies the parent business-level application for the composition unit.

backingID (read-only)
Specifies the composition unit source ID.

name (read-only)
Specifies the name of the composition unit.

description
Specifies a description of the composition unit.

startingWeight
Specifies the starting weight of the composition unit.

startedOnDistributed
Specifies whether to start the composition unit after distributing changes to the target nodes. The default value is `false`.

restartBehaviorOnUpdate
Specifies the nodes to restart after editing the composition unit. Specify `ALL` to restart each target node. Specify `DEFAULT` to restart the nodes controlled by the sync plug-ins. Specify `NONE` to prevent the system from restarting nodes.

For example, specify the syntax for this step as `-CUOptions [[.* .* cu4 "cu4 description" 0 false DEFAULT]]`

-MapTargets
Specifies additional properties for the composition unit target mapping. Specify the following options with the MapTargets step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

server Specifies the target or targets to deploy the composition units. The default value is the `server1` server. Use the plus sign character (+) to specify multiple targets. Use the plus sign character (+) as a prefix to add an additional target. Specify the complete object name format for each server that is not a WebSphere Application Server server.

For example, specify the syntax of this step as `-MapTargets [[a1.jar cluster1+cluster2] [a2.jar server1+server2]]`

-ActivationPlanOptions
Specifies additional properties for the composition unit activation plan. Specify the following options with the ActivationPlanOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

activationPlan
Specifies a list of components as the activation plan. Specify each activation plan in the format `specName=xxx,specVersion=yyy`, where `specName` represents the name of the specification and is required. Use the plus sign character (+) to specify multiple activation plans.

For example, specify the syntax of this step as `-ActivationPlanOptions [[a1.jar specname=actplan0+actplan1] [a2.jar specname=actplan1+specname=actplan2]]`

-RelationshipOptions
Specifies additional properties for relationships between assets, composition units, and business-level applications. Use this step if the source ID of the composition unit is an asset that has a matching composition unit in the business-level application. Specify the following options with the RelationshipOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI). The default value is `default`, which deploys the entire composition unit.

relationship
Defines the composition unit relationships. Specify the composition unit object name in the format: `cuName=xxx`. Use the plus sign character (+) to specify multiple composition unit object names in the relationship. If the composition unit specified in the relationship does not exist under the same business-level application, the system removes the composition unit from the relationship.

matchTarget
Specifies whether to match the targets of the composition unit relationship with the targets of the new composition unit. The default value is `true`.

For example, specify the syntax of this step as `–RelationshipOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

Return value

The command returns the configuration ID of the composition unit that the system edits.

Batch mode example usage

Use the following examples to edit a composition unit of an asset and replace the target from existing targets:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID myBLA –cuID cu1 –CUOptions [[.* .* cu1 cudesc 1 false DEFAULT]] -MapTargets
  [[.* server2]] -ActivationPlanOptions [.* #specname=actplan0+specname=actplan2]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'myBLA', '–cuID', 'cu1', '–CUOptions',
  '[[.* .* cu1 cudesc 1 false DEFAULT]]', '-MapTargets', '
  [[.* server2]]', '-ActivationPlanOptions', '[.* #specname=actplan0+specname=actplan2]'])
```

Use the following examples to edit a composition unit of an asset and its relationships:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID ourBLA –cuID cu4 –CUOptions [[.* .* cu4 "new cu desc" 1 false DEFAULT]]
   –MapTargets [[a1.jar server1+server2]] –RelationshipOptions [[a1.jar cuname=cub true]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'ourBLA', '–cuID', 'cu4', '–CUOptions', '
  [[.* .* cu4 "new cu desc" 1 false DEFAULT]]', '–MapTargets', '[[a1.jar server1+server2]]', '–RelationshipOptions',
  '[[a1.jar cuname=cub true]]'])
```

Use the following examples to edit a composition unit by adding a new relationship to the existing relationship:

- Using Jython string:

```
AdminTask.editCompUnit('[-blaID ourBLA –cuID cu4 –CUOptions [[.* .* cu4 "new cu desc" 1 false DEFAULT]]
   –MapTargets [[a1.jar server1+server2]] –RelationshipOptions [[a1.jar +cuname=cuc true]] -ActivationPlanOptions
  [a1.jar +specname=actplan2#specname=actplan1]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'ourBLA', '–cuID', 'cu4', '–CUOptions', '
  [[.* .* cu4 "new cu desc" 1 false DEFAULT]]', '–MapTargets', '[[a1.jar server1+server2]]', '–RelationshipOptions',
   '[[a1.jar +cuname=cuc true]]', '-ActivationPlanOptions', '[a1.jar +specname=actplan2#specname=actplan1]'])
```

Use the following examples to edit an enterprise composition unit configuration:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID yourBLA –cuID defaultapp –MapModulesToServers
  [[defaultapp.war .* WebSphere:cluster=cluster1][Increment.jar .* Websphere:cluster=cluster2]]
   –CtxRootForWebMod [[defaultapp.war .* /]] –MapWebModToVH [[defaultapp.war .* vh1]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'yourBLA', '–cuID', 'defaultapp', '–MapModulesToServers',
  '[[defaultapp.war .* WebSphere:cluster=cluster1][Increment.jar .* Websphere:cluster=cluster2]]', '–CtxRootForWebMod',
  '[[defaultapp.war .* /]]', '–MapWebModToVH', '[[defaultapp.war .* vh1]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editCompUnit('-interactive')
```

listCompUnits

The listCompUnits command displays each composition unit that is associated with a specific business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional parameters

-includeDescription
Specifies whether to include a description of each asset that the command returns. (String, optional)

-includeType
Specifies whether to include the type for each asset that the command returns. (String, optional)

Return value

The command returns a list of configuration IDs and the type for each composition unit, as the following example displays:

```
Websphere:cuname=cu1
asset
"description for cu1"
Websphere:cuname=cu4
bla
"description for cu4"
WebSphere:cuname=defaultapp
enterprise
"description for defaultapp"
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listCompUnits('-blaID blaname=theirBLA')
```

- Using Jython list:

```
AdminTask.listCompUnits(['-blaID', 'blaname=theirBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listCompUnits('-interactive')
```

setCompUnitTargetAutoStart

The setCompUnitTargetAutoStart command enables or disables automatic starting of composition units. If you enable this option, the system automatically starts the composition unit when the composition unit target starts.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete configuration ID if the system matches it to a unique business-level application ID. (String, required)

-cuID
Specifies the composition unit of interest. The command accepts an incomplete configuration ID if the system matches it to a unique composition unit ID. (String, required)

-targetID
Specifies the name of the target of interest. For example, specify the server name to set the target to a specific server. (String, required)

-enable
Specifies whether to automatically start the composition unit of interest when the specified target starts. Specify `true` to start the composition unit automatically. If you do not specify `true`, the system will not start the composition unit when the target starts. The default value is `true`. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setCompUnitTargetAutoStart('-blaID bla1 –cuID cu1 –targetID server1 –enable true')
```

- Using Jython list:

```
AdminTask.setCompUnitTargetAutoStart(['-blaID', 'bla1', '–cuID', 'cu1', '–targetID',
   'server1', '–enable', 'true'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.setCompUnitTargetAutoStart('-interactive')
```

viewCompUnit

The viewCompUnit command displays configuration information for a composition unit that belongs to a specific business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. This parameter accepts an incomplete configuration ID if the system matches it to a unique business-level application ID. (String, required)

-cuID
Specifies the configuration ID of the composition unit of interest. This parameter accepts an incomplete configuration ID if the system matches it to a unique composition unit ID. (String, required)

Optional parameters

None

Return value

The command returns configuration information for the composition unit of interest, as the following example displays:

```
Specify Composition Unit options (CUOptions)

Specify name, description options for Composition Unit.

Parent BLA (parentBLA): [WebSphere:blaname=myBLA]
Backing Id (backingId): [WebSphere:assetname=asset1.zip]
Name (name): [cu1]
Description (description): [cuDesc]
Starting Weight (startingWeight): [0]
Started on distributed (startedOnDistributed): [false]
Restart behavior on update (restartBehaviorOnUpdate): [DEFAULT]

Specify servers (MapTargets)

Specify targets such as application servers or clusters of application servers where you want to deploy the cu contained in the application.

Deployable Unit (deplUnit): [default]
*Servers (server): [WebSphere:node=myNode,server=server1]

Specify Composition Unit activation plan options (ActivationPlanOptions)

Specify CU activation plan optionsDeployableUnit Name (deplUnit): [default]
Activation Plan (activationPlan): [WebSphere:specname=actplan0+WebSphere:specname=actplan1]
```

Batch mode example usage

The following example displays configuration information for a non-enterprise asset:

- Using Jython string:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID myCompUnit')
```

- Using Jython list:

```
AdminTask.viewCompUnit(['-blaID', 'myBLA', '-cuID', 'myCompUnit'])
```

The following example displays configuration information for an enterprise asset:

- Using Jython string:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID defaultApplication')
```

- Using Jython list:

```
AdminTask.viewCompUnit(['-blaID', 'myBLA', '-cuID', 'defaultApplication'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewCompUnit('-interactive')
```

createEmptyBLA

The createEmptyBLA command to create an empty business-level application. After creating a business-level application, you can add assets or other business-level applications as composition units to the application.

Target object

None

Required parameters

-name
Specifies a unique name for the new business-level application. (String, required)

Optional parameters

-description
Specifies a description of the new business-level application. (String, optional)

Return value

The command returns the configuration ID of the new business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createEmptyBLA('-name myBLA -description "my description for BLA"')
```

- Using Jython list:

```
AdminTask.createEmptyBLA(['-name', 'myBLA', '-description', '"my description for BLA"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createEmptyBLA('-interactive')
```

deleteBLA

The deleteBLA command removes a business-level application from your configuration. Before deleting a business-level application, use the deleteCompUnit command to remove each configuration unit that is associated with the business-level application. Also, verify that no other business-level applications reference the business-level application to delete.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete ID for the blaID parameter, as long as the system can match the string to a unique identifier. For example, you can specify the `myBLA` partial ID to identify the `WebSphere:blaname=myBLA` configuration ID. (String, required)

Optional parameters

None

Return value

The command returns the configuration ID of the deleted business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.deleteBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteBLA('-interactive')
```

editBLA

The editBLA command modifies the description of a business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-BLAOptions
Use the BLAOptions step to specify a new description for the business-level application of interest.

name (read-only)
Specifies the name of the business-level application.

description
Specifies a description of the business-level application.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.editBLA('-blaID DefaultApplication –BLAOptions [[.* "my new description"]]')
```

- Using Jython list:

```
AdminTask.editBLA(['-blaID', 'DefaultApplication', '–BLAOptions', '[[.* "my new description"]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editBLA('-interactive')
```

getBLAStatus

The getBLAStatus command displays whether a business-level application or composition unit is running or stopped.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. Use the listBLAs command to display a list of business-level application configuration IDs. (String, required)

Optional parameters

-cuID
Specifies the configuration ID of the composition unit of interest. Use the listCompUnits command to display a list of composition unit configuration IDs. (String, optional)

Return value

The command returns the status of the business-level application or composition unit of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getBLAStatus('-blaID WebSphere:blaname=myBLA -cuID Websphere:cuname=cu1')
```

- Using Jython list:

```
AdminTask.getBLAStatus(['-blaID', 'WebSphere:blaname=myBLA', '-cuID', 'Websphere:cuname=cu1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getBLAStatus('-interactive')
```

listBLAs

The listBLAs command displays the business-level applications in your configuration.

Target object

None

Optional parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, optional)

-includeDescription
Specifies whether to include a description of each business-level application that the command returns. Specify `true` to display the business-level application descriptions. (String, optional)

Return value

The command returns a list of configuration IDs for each business-level application in your configuration, as the following example displays:

```
WebSphere:blaname=myBLA
WebSphere:blaname=yourBLA
```

Batch mode example usage

The following example lists each business-level application in the configuration:

- Using Jython:

```
AdminTask.listBLAs()
```

Use the following examples to list each business-level application in the configuration:

- Using Jython string:

```
AdminTask.listBLAs('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.listBLAs(['-blaID', 'myBLA'])
```

Use the following examples to list each business-level application and the corresponding descriptions:

- Using Jython string:

```
AdminTask.listBLAs('-includeDescription true')
```

- Using Jython list:

```
AdminTask.listBLAs(['-includeDescription', 'true'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.listBLAs('-interactive')
```

listControlOps

The listControlOps command displays the control operations for a business-level application and the corresponding composition units.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional parameters

-cuID
Specifies the composition unit of interest. (String, optional)

-opName
Specifies the operation name of interest. (String, optional)

-long
Specifies whether to include additional configuration information in the command output. (String, optional)

Return value

The command returns a list of operations, operation descriptions, and parameter descriptions for the query scope, as the following example displays:

```
"Operation: start"
"   Description: Start operation"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"Operation: stop"
"   Description: Stop operation"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"Operation: clearCache"
"   Description: Clears specified cache or all caches"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"   Parameter: cacheName"
"      Description: Name of cache to clear.  If not specified, all caches are cleared."
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listControlOps('-blaID myBLA –cuID myservice.jar –long true')
```

- Using Jython list:

```
AdminTask.listControlOps(['-blaID', 'myBLA', '–cuID', 'myservice.jar', '–long true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listControlOps('-interactive')
```

startBLA

The startBLA command starts the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application to start. The command accepts an incomplete configuration ID if the system matches the string to a unique ID in your configuration. (String, required)

Return value

The command returns a status message if the business-level application starts. If the business-level application does not start, the command does not return output. The following example displays the status message output:

```
BLA ID of started BLA if the BLA was not already running.
```

Batch mode example usage

- Using Jython string:

```
AdminTask.startBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.startBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.startBLA('-interactive')
```

stopBLA

The stopBLA command stops the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application to stop. The command accepts an incomplete configuration ID if the system matches the string to a unique ID in your configuration. (String, required)

Return value

The command returns a status message if the business-level application stops. If the business-level application does not stop, the command does not return output. The following example displays the status message output:

```
BLA ID of stopped BLA if the BLA was not already stopped.
```

Batch mode example usage

- Using Jython string:

```
AdminTask.stopBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.stopBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.stopBLA('-interactive')
```

viewBLA

The viewBLA command displays the name and description of the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete configuration ID if the system matches the string to a unique business-level application. (String, required)

Optional parameters

None

Return value

The command returns configuration information for the business-level application of interest, as the following example displays:

```
Specify BLA options (BLAOptions)

Specify options for BLA

*BLA Name (name): [DefaultApplication]
BLA Description (description): []
```

Batch mode example usage

- Using Jython string:

```
AdminTask.viewBLA('-blaID DefaultApplication')
```

- Using Jython list:

```
AdminTask.viewBLA(['-blaID', 'DefaultApplication'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewBLA('-interactive')
```

JSFCommands command group for the AdminTask object

You can use the Jython scripting language to display and modify the JavaServer Faces (JSF) implementation.

Use the following command to administer JSF:

- "listJSFImplementation"
- "modifyJSFImplementation" on page 243

listJSFImplementation

The listJSFImplementation command displays the JSF implementation and version for a specific application.

Target object

Specify the name of the application of interest. (String, required)

Required parameters

None.

Return value

The command displays the JSF implementation and version. For example, if the command returns "SUNRI1.2", then JSF uses version 1.2 of the Sun Reference Implementation.

Batch mode example usage

- Using Jython string:

```
AdminTask.listJSFImplementation('application1')
```

- Using Jython list:

```
AdminTask.listJSFImplementation('application1')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listJSFImplementation('-interactive')
```

modifyJSFImplementation

The modifyJSFImplementation command modifies the JSF implementation for a specific application.

Target object

Specify the name of the application of interest. (String, required)

Required parameters

-implName
Specifies the name of the implementation to use. Specify SUNRI1.2 to use the Sun Reference 1.2 Implementation, or specify MyFaces1.2 to use the Apache MyFaces 1.2 project implementation. By default, applications use the Sun Reference Implementation. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyJSFImplementation('-implName MyFaces1.2')
```

- Using Jython list:

```
AdminTask.modifyJSFImplementation('-implName', 'MyFaces1.2')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyJSFImplementation('-interactive')
```

Application management command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage applications with the wsadmin tool. Use the commands and parameters in the AppManagementCommands group can be used to display and process SQL-Java (SQLJ) profiles or pureQuery bind files.

The AppManagementCommands command group for the AdminTask object includes the following commands:

listSqljProfiles

The listSqljProfiles command parses the .ear file of the specified application and returns a list of .ser files found. SQLJ profiles have a .ser filename extension. If there are any files in the .ear file that are not SQLJ profiles, but have a .ser filename extension, those files may be listed also.

Parameters and return values

-appName
The name of the installed application. Your application must be installed prior to running customization and binding on it. This parameter is required.

Examples

Batch mode example usage:

- Using JACL:

```
$AdminTask listSqljProfiles {-appName application_name}
```

- Using Jython:

```
print AdminTask.listSqljProfiles('-appName application_name')
```

Interactive mode example usage:

- Using JACL:

```
$AdminTask listSqljProfiles -interactive
```

- Using Jython:

```
print AdminTask.listSqljProfiles('-interactive')
```

Output appears with syntax specific to the local operating system. The list of available profiles can be added to a group file .grp directly.

processSqljProfiles

The processSqljProfiles command creates a DB2 customization of the SQLJ profiles. The command optionally, by default, calls the SQLJ profile binder to bind the DB2 packages.

Note: If you are processing a large enterprise application, or you are processing many SQLJ profiles, the process might take longer than the default timeout for the wsadmin tool. The default connection timeout for the wsadmin tool is set to three minutes. If the default timeout is reached and you lose the connection to the server, the wsadmin console issues a timeout statement. You can check the system output log for the final results of the customization and bind process and the amount of time for that the process. Do not execute the processSqljProfiles command again until the previous command has completed, or the results may be unpredictable.

To prevent this disconnection, configure the session timeout to a longer period of time. See the system output log for the total processing time, and use that time period as a basis for the new timeout value. To extend the default timeout value, change the wsadmin properties file that corresponds to the connection type that you are using:

- For the SOAP connection type, change the following entry in the soap.client.props file:

```
com.ibm.SOAP.requestTimeout=180
```

- For JSR160RMI and RMI connection types, change the following entry in the sas.client.props file:

```
com.ibm.CORBA.requestTimeout=180
```

- For the IPC connection type, change the following entry in the ipc.client.props file:

```
com.ibm.IPC.requestTimeout=180
```

There are two ways you can verify whether the binding or customization took place:

- If you performed a customization process, you can run a query from the command line to see the application .ear files that were changed:

```
wsadmin>print AdminConfig.hasChanges()
```

  The query will return 0 if there are no changes, and 1 if changes occurred on the server. To view the configuration files that have unsaved changes, run:

```
wsadmin>print AdminConfig.queryChanges()
```

  .
- View the System Out log to determine if the binding or processing was successful.

Target object

The installed application SQLJ profiles. These profiles are either single, serial .ser files or profiles grouped in a .grp group file. This target object is required.

Parameters and return values

-appName
The name of the installed application. Your application must be installed prior to running customization and binding on it. This parameter is required.

-classpath
The path that tells the application server where to find the necessary SQLJ driver .jar files. This parameter is optional.

-dburl
The location of the DB2 server on the network. This parameter is optional.

-user
User name of the account performing the access to the DB2 database. This parameter is optional.

-password
Password for the account accessing the DB2 database. This parameter is optional.

-options
Additional options that are used with the **db2sqljcustomize** command may be inserted under the **-options** parameter except for the parameters listed above. This parameter is optional. For additional information about the **db2sqljcustomize** command, consult db2sqljcustomize - SQLJ profile customizer.

-profiles
The location of the SQLJ profiles .ser files or .grp file. This parameter is required.

Examples

Batch mode example usage:

Interactive mode example usage:

```
wsadmin>print AdminTask.processSqljProfiles('-interactive') Process serialized SQLJ
 profiles. Process the serialized SQLJ profiles in an installed application.  Customize the profiles with run time information and
 bind static SQL packages in a database.   Refer to the Database SQLJ customize and bind documentation.    Do only bind
 processing. (bindOnly): false  *Application name. (appName): Application  Classpath to SQLJ tools. (classpath):
 C:/IBM/SQLLIB/java/db2jcc.jar  Database connection URL. (dbURL):  Database connection user name. (user):  Database connection
 password. (password):  Options for SQLJ tools. (options):  *SQLJ profile names. (profiles): c:\\temp\\ApplicationSerNames.grp
 Process serialized SQLJ profiles.    F (Finish) C (Cancel)   Select [F, C]: [F]  WASX7278I: Generated command line:
 AdminTask.processSqljProfiles('[-bindOnly false -appName Application -classpath [C:/IBM/SQLLIB/java/db2jcc.jar] -profiles
 [c:\\temp\\ApplicationSerNames.grp ]]')
```

listPureQueryBindFiles

The listPureQueryBindFiles command parses the .ear file of the specified application and returns a list of .bindprops and .pdqxml files found. PureQuery bind options files have a .bindprops filename extension. Bind files have a .pdqxml filename extension. If .ear file contains files that are not pureQuery bind files but have a .bindprops or a .pdqxml filename extension, those files may also be listed.

Parameters and return values

-appName
The name of the installed application. This parameter is required.

Examples

Batch mode example usage:

- Using JACL:

```
$AdminTask listPureQueryBindFiles {-appName application_name}
```

- Using Jython:

```
print AdminTask.listPureQueryBindFiles('-appName application_name')
```

Interactive mode example usage:

- Using JACL:

```
$AdminTask listPureQueryBindFiles -interactive
```

- Using Jython:

```
print AdminTask.listPureQueryBindFiles('-interactive')
```

Output appears with syntax specific to the local operating system.

processPureQueryBindFiles

The processPureQueryBindFiles command invokes the DB2 pureQuery bind utility on a list of pureQuery bind files.

Note: If you are processing a large enterprise application, or you are processing many pureQuery bind files using wsadmin, the process might take longer than the default timeout for the wsadmin tool. The default connection timeout for the wsadmin tool is set to three minutes. If the default timeout is reached and the process running on the server has not yet completed, the wsadmin console issues a timeout statement. You can check the system output log on the server for the final results of the bind process and the time when that process completed. Do not execute the processPureQueryBindFiles command again until the previous command has completed, or the results may be unpredictable.

To prevent this timeout, configure the wsadmin request timeout to a longer period of time. After a successful customization and binding process, use the system output log to estimate the total processing time. Use this time period as a basis for the new timeout value. To extend the default timeout value, change the wsadmin properties file that corresponds to the connection type that you are using:

- For the SOAP connection type, change the following entry in the soap.client.props file:

```
com.ibm.SOAP.requestTimeout=180
```

- For JSR160RMI and RMI connection types, change the following entry in the sas.client.props file:

```
com.ibm.CORBA.requestTimeout=180
```

- For the IPC connection type, change the following entry in the ipc.client.props file:

```
com.ibm.IPC.requestTimeout=180
```

To verify whether the binding took place, view the System Out log to determine if the bind processing was successful.

Parameters and return values

-appName

The name of an installed application that contains the pureQuery bind files to be processed. Your application must be installed prior to running binding on it.

-classpath

A list of the paths to the Java archive (JAR) files that contain the pureQuery bind utility and its dependencies: pdq.jar, pdqmgmt.jar, db2jcc4.jar or db2jcc.jar, db2jcc_license_cisuz.jar or db2jcc_license_cu.jar. Use / or \\\\ as a file separator. Use a blank space to separate the paths for the JAR files.

-dburl

The URL for connecting to the database. The format is jdbc:db2://*server_name*:*port*/*database_name*.

-user

User name of the account performing the access to the DB2 database.

-password

Password for the account accessing the DB2 database.

-options

Any additional options that are needed by the pureQuery bind utility. Provide bind options as **-bindoptions** ″**bind_options_string**″. For additional information about the pureQuery bind utility, consult the topic on the pureQuery Bind utility.

-files

A list of the names of the pureQuery bind files to be processed. The bind file path names must be relative to the application .ear file that contains them. Use / or \\\\ as a file separator. If you specify multiple profile paths, use a blank space to separate them.

Examples

Batch mode example usage:

Interactive mode example usage:

```
print AdminTask.processPureQueryBindFiles('-interactive') Process pureQuery bind files.
 Process the pureQuery bind files in an installed application.  Bind static SQL packages in a database.  Refer to IBM pureQuery
 Bind utility documentation.  *Application name. (appName): MyApp Classpath to pureQuery Bind utility. (classpath):
 /pdq_home/pdq.jar /pdq_home/pdqmgmt.jar /db2_home/SQLLIB/java/db2jcc4.jar /db2_home/SQLLIB/java/db2jcc_license_cu.jar *Database
 connection URL. (url): jdbc:db2://hostname:50000/databasename Database connection user name. (user): dbuser1 Database connection
 password. (password): dbpswrd1 Options for the pureQuery Bind utility. (options): -bindoptions "BLOCKING NO" *pureQuery bind file
 names. (files): META-INF/xyz.bindprops META-INF/abc.bindprops  Process pureQuery bind files.  F (Finish) C (Cancel)  Select [F,
 C]: [F] WASX7278I: Generated command line: AdminTask.processPureQueryBindFiles('[-appName MyApp -classpath [/pdq_home/pdq.jar
 /pdq_home/pdqmgmt.jar /db2_home/SQLLIB/java/db2jcc4.jar /db2_home/SQLLIB/java/db2jcc_license_cu.jar ] -url
 jdbc:db2://hostname:50000/databasename -user dbuser1 -password ***** -options [-bindoptions "BLOCKING NO"] -files
 [META-INF/xyz.bindprops META-INF/abc.bindprops ]]')
```

Related concepts

Changing the console session expiration
Run this JACL script to set how long Integrated Solutions Console can be used until the login session expires.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Customizing and binding profiles for Structured Query Language in Java (SQLJ) applications
Simplify the process of customizing and binding SQLJ profiles for your applications by performing these functions in the administrative console or with scripting. SQLJ profiles must be customized and bound before the enterprise application can use the application's embedded SQL.

Task overview: Data Studio pureQuery
Data Studio pureQuery provides Java Persistence API (JPA) users an alternative way to access a DB2 database. PureQuery supports static Structured Query Language (SQL).

db2sqljcustomize - SQLJ profile customizer

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

"Wsadmin tool" on page 1181
The wsadmin tool runs scripts. You can use the wsadmin tool to manage application server as well as the configuration, application deployment, and server runtime operations.

Related information

pureQuery Bind utility

Chapter 5. Managing deployed applications using scripting

Use these topics to learn more about managing deployed applications with scripting and the wsadmin tool.

- Start enterprise applications and stop enterprise applications. You can use the AdminControl object to start an application that is not running (has a status of Stopped) or stop an application that is running (has a status of Started).
- Start business-level applications and stop business-level applications. You can use the wsadmin tool and the BLAManagement command group to start and stop business-level applications.
- Update applications. Use the wsadmin tool to update installed applications on an application server.
- Manage assets. Use the commands in the BLAManagement command group to manage your asset configuration. This topic provides examples for listing assets, viewing asset configuration data, removing assets from the asset repository, updating one or more files for assets, and exporting assets.
- Manage composition units. Use the commands in the BLAManagement command group to manage composition units. This topic provides examples for adding, removing, editing, exporting, and viewing composition units.
- List application modules. Use the AdminApp object listModules command to list the modules in an installed application.
- Query the application state. Use the wsadmin tool and scripting to determine if an application is running.
- Configure session management for applications or configure session management for web modules. Use scripting and the wsadmin tool to configure applications for session management in applications or Web modules.
- Configure a shared library for an application server or configure a shared library for an application. You can use scripting to configure a shared library for application servers or applications.
- Set background applications. You can enable or disable a background application using scripting and the wsadmin tool.
- Configure name space bindings. Use this topic to configure name space bindings with the Jython or Jacl scripting languages and the wsadmin tool.
- Export applications. You can export your applications before you update installed applications or before you migrate to a different version of the product.

Starting applications with scripting

Use scripting and the wsadmin tool to start an application that is not running.

Before you begin

There are two ways to complete this task. This topic uses the AdminControl object to start an application. Alternatively, you can use the scripts in the AdminApplication script library to start, stop, and manage applications.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application manager MBean for the server where the application resides and assign it the appManager variable. The following example returns the name of the application manager MBean.
   - Using Jacl:

     ```
     set appManager [$AdminControl queryNames cell=mycell,node=mynode,type=ApplicationManager,process=server1,*]
     ```

   - Using Jython:

     ```
     appManager = AdminControl.queryNames('cell=mycell,node=mynode,type=ApplicationManager,process=server1,*')
     print appManager
     ```

   where:

`set`	is a Jacl command

`appManager`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`queryNames`	is an AdminControl command
`cell=mycell,node=mynode,type=ApplicationManager,process=server1`	is the hierarchical containment path of the configuration object
`print`	is a Jython command

Example output:

```
WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=ApplicationManager,
type=ApplicationManager,node=mynode,process=server1
```

3. Start the application. The following example invokes the startApplication operation on the MBean, providing the application name that you want to start.
   - Using Jacl:

     ```
     $AdminControl invoke $appManager startApplication myApplication
     ```

   - Using Jython:

     ```
     AdminControl.invoke(appManager, 'startApplication', 'myApplication')
     ```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`invoke`	is an AdminControl command
`appManager`	evaluates to the ID of the server that is specified in step number 1
`startApplication`	is an attribute of the modify command
`myApplication`	is the value of the startApplication attribute

Starting business-level applications using scripting

You can use the wsadmin tool and the BLAManagement command group to start business-level applications.

Before you begin

There are two ways to complete this task. Use the BLAManagement command group for the AdminTask object or the scripts in the AdminBLA script library to start your business-level applications.

- Use the AdminTask object commands to start business-level applications.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List the business-level applications in your environment.

     Use the listBLAs command to display a list of business-level applications in your environment, as the following example demonstrates:

```
AdminTask.listBLAs()
```

     You can optionally specify the partial name of the business-level application of interest to display the configuration ID of the business-level application. The command accepts a partial business-level

application name if the system matches the specified name to a unique configuration ID. Use the following example to set the configuration ID of the myBLA business-level application to the blaID variable:

```
myBLA=AdminTask.listBLAs('-blaID BLA1')
```

3. Determine the status of the business-level application.

   Use the getBLAStatus command to display the status of the business-level application of interest, as the following example demonstrates:

```
AdminTask.getBLAStatus('-blaID myBLA')
```

   The command returns the status of the business-level application as STOPPED or STARTED.

4. Start the business-level application.

   Use the startBLA command to start the business-level application, as the following example demonstrates:

```
AdminTask.startBLA('-blaID myBLA')
```

   The command returns the following message if the system successfully starts the business-level application:

```
BLA ID of started BLA if the BLA was not already running.
```

- Use the Jython script library to start business-level applications.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List the business-level applications in your environment.

     Use the listBLAs script to display a list of business-level applications in your environment, using the following syntax:

```
AdminBLA.listBLAs(blaName, displayDescription)
```

     You can specify one, both, or neither the blaName and displayDescription arguments. Use the blaName argument to specify the name of a specific business-level application, and the displayDescription argument to specify whether to display the description of each returned business-level application. Specify an empty string in place of arguments that you do not want to specify, as the following example demonstrates:

```
AdminBLA.listBLAs("", "true")
```

  3. Start the business-level application.

     Use the startBLA script to start the business-level application, using the following syntax:

```
AdminBLA.startBLA(blaName)
```

     Use the blaName argument to specify the name of the business-level application to start, as the following example demonstrates:

```
AdminBLA.startBLA("myBLA")
```

Stopping applications with scripting

You can use the wsadmin tool and the AdminConfig object to stop applications.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminControl object to stop the application. Alternatively, you can use the scripts in the AdminApplication script library to start, stop, and administer your application configurations.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application manager MBean for the server where the application resides, and assign it to the appManager variable.
   - Using Jacl:

     ```
     set appManager [$AdminControl queryNames cell=mycell,node=mynode,type=
     ApplicationManager,process=server1,*]
     ```

   - Using Jython:

```
appManager = AdminControl.queryNames('cell=mycell,node=mynode,type=
ApplicationManager,process=server1,*')
print appManager
```

where:

`set`	is a Jacl command
`appManager`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`queryNames`	is an AdminControl command
`cell=mycell,node=mynode,type=ApplicationManager,process=server1`	is the hierarchical containment path of the configuration object
`print`	is a Jython command

This command returns the application manager MBean.

Example output:

```
WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=ApplicationManager,
type=ApplicationManager,node=mynode,process=server1
```

3. Query the running applications belonging to this server and assign the result to the apps variable.
   - Using Jacl:

     ```
     set apps [$AdminControl queryNames cell=mycell,node=mynode,type=Application,
     process=server1,*]
     ```

   - Using Jython:

     ```
     # get line separator
     import  java.lang.System  as sys
     lineSeparator = sys.getProperty('line.separator')

     apps = AdminControl.queryNames('cell=mycell,node=mynode,type=Application,
     process=server1,*').split(lineSeparator)
     print apps
     ```

   where:

`set`	is a Jacl command
`apps`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`queryNames`	is an AdminControl command
`cell=mycell,node=mynode,type=ApplicationManager,process=server1`	is the hierarchical containment path of the configuration object
`print`	is a Jython command

This command returns a list of application MBeans.

Example output:

```
WebSphere:cell=mycell,name=adminconsole,mbeanIdentifier=deployment.xml
#ApplicationDeployment_1,type=Application,node=mynode,Server=server1,
process=server1,J2EEName=adminconsole
WebSphere:cell=mycell,name=filetransfer,mbeanIdentifier=deployment.xml
#ApplicationDeployment_1,type=Application,node=mynode,Server=server1,
process=server1,J2EEName=filetransfer
```

4. Stop all the running applications.
   - Using Jacl:

     ```
     foreach app $apps {
          set appName [$AdminControl getAttribute $app name]
          $AdminControl invoke $appManager stopApplication $appName}
     ```

   - Using Jython:

     ```
     for app in apps:
         appName = AdminControl.getAttribute(app, 'name')
         AdminControl.invoke(appManager, 'stopApplication', appName)
     ```

   This command stops all the running applications by invoking the stopApplication operation on the MBean, passing in the application name to stop.

Results

Once you complete the steps for this task, all running applications on the server are stopped.

Stopping business-level applications with scripting

You can use the wsadmin tool and the BLAManagement command group to stop business-level applications.

Before you begin

There are two ways to complete this task. Use the BLAManagement command group for the AdminTask object or the scripts in the AdminBLA script library to stop your business-level applications.

- Use the AdminTask object to stop business-level applications.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List the business-level applications in your environment.

     Use the listBLAs command to display a list of business-level applications in your environment, as the following example demonstrates:

     ```
     AdminTask.listBLAs()
     ```

     You can optionally specify the partial name of the business-level application of interest to display the configuration ID of the business-level application. The command accepts a partial business-level application name if the system matches the specified name to a unique configuration ID. Use the following example to set the configuration ID of the `myBLA` business-level application to the `blaID` variable:

     ```
     myBLA=AdminTask.listBLAs('-blaID BLA1')
     ```

  3. Determine the status of the business-level application.

     Use the getBLAStatus command to display the status of the business-level application of interest, as the following example demonstrates:

     ```
     AdminTask.getBLAStatus('-blaID myBLA')
     ```

     The command returns the status of the business-level application as `STOPPED` or `RUNNING`.
  4. Stop the running business-level application.

     Use the stopBLA command to stop the business-level application, as the following example demonstrates:

     ```
     AdminTask.stopBLA('-blaID myBLA')
     ```

     The command returns the following message if the system successfully stops the business-level application:

     ```
     BLA ID of stopped BLA if the BLA was not already stopped.
     ```

- Use the Jython script library to stop business-level applications.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List the business-level applications in your environment.

Use the listBLAs script to display a list of business-level applications in your environment, using the following syntax:

```
AdminBLA.listBLAs(blaName, displayDescription)
```

You can specify one, both, or neither the blaName and displayDescription arguments. Use the blaName argument to specify the name of a specific business-level application, and the displayDescription argument to specify whether to display the description of each returned business-level application. Specify an empty string in place of arguments that you do not want to specify, as the following example demonstrates:

```
AdminBLA.listBLAs("", "true")
```

3. Stop the business-level application.

   Use the stopBLA script to stop the business-level application, using the following syntax:

```
AdminBLA.stopBLA(blaName)
```

Use the blaName argument to specify the name of the business-level application to stop, as the following example demonstrates:

```
AdminBLA.stopBLA("myBLA")
```

Updating installed applications with the wsadmin tool

Use the wsadmin tool and scripting to update installed applications on an application server.

About this task

Both the **update** command and the **updateinteractive** command support a set of options. You can also obtain a list of supported options for an Enterprise Archive (EAR) file using the **options** command, for example:

Using Jacl:

```
$AdminApp options
```

Using Jython:

```
print AdminApp.options()
```

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Perform the following steps to update an application:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Update the installed application using one of the following options:
   - The following command updates a single file in a deployed application:
     – Using Jacl:

```
$AdminApp update app1 file {-operation update -contents
 /home/myProfile/apps/app1/my.xml -contenturi app1.jar/my.xml}
```

     – Using Jython string:

```
AdminApp.update('app1', 'file', '[-operation update -contents /home/myProfile/
 apps/app1/my.xml -contenturi app1.jar/my.xml]')
```

     – Using Jython list:

```
AdminApp.update('app1', 'file', ['-operation', 'update', '-contents',
 '/home/myProfile/apps/app1/my.xml', '-contenturi', 'app1.jar/my.xml'])
```

where:

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
file	is the content type value
operation	is an option of the **update** command
update	is the value of the operation option
contents	is an option of the **update** command
/apps/app1/my.xml	is the value of the contents option
contenturi	is an option of the **update** command
app1.jar/my.xml	is the value of the contenturi option

- The following command adds a module to the deployed application, if the module does not exist. Otherwise, the existing module is updated.
  - Using Jacl:

```
$AdminApp update app1 modulefile {-operation addupdate -contents
 /home/myProfile/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb  -BindJndiForEJBNonMessageBinding {{"Increment EJB
 module"  Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
```

  - Using Jython string:

```
AdminApp.update('app1', 'modulefile', '[-operation addupdate -contents
 /home/myProfile/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment EJB
 module" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

  - Using Jython list:

```
bindJndiForEJBValue = [["Increment EJB module", "Increment", "
 Increment.jar,META-INF/ejb-jar.xml", "Inc"]]  AdminApp.update('app1', 'modulefile', ['-operation', 'addupdate', '-contents',
 '/home/myProfile/apps/app1/Increment.jar', '-contenturi','Increment.jar' '-nodeployejb', `-BindJndiForEJBNonMessageBinding',
 bindJndiForEJBValue])
```

where:

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
modulefile	is the content type value
operation	is an option of the **update** command
addupdate	is the value of the operation option
contents	is an option of the **update** command
/apps/app1/Increment.jar	is the value of the contents option
contenturi	is an option of the **update** command
Increment.jar	is the value of the contenturi option
nodeployejb	is an option of the **update** command
BindJndiForEJBNonMessageBinding	is an option of the **update** command

"Increment EJB module" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option. The value of this option is defined in your application configuration. To determine the value of this option, use the following Jython or Jacl command:
	
	Using Jython:
	`AdminApp.view('myAppName')`
	
	Using Jacl:
	`$AdminApp view myAppName`

bindJndiForEJBValue	is a Jython variable that contains the value of the BindJndiForEJBNonMessageBinding option

- The following command uses a partial application to update a deployed application:
  - Using Jacl:

```
$AdminApp update app1 partialapp {-contents
 /home/myProfile/apps/app1/app1Partial.zip}
```

  - Using Jython string:

```
AdminApp.update('app1', 'partialapp', '[-contents
 /home/myProfile/apps/app1/app1Partial.zip]')
```

  - Using Jython list:

```
AdminApp.update('app1', 'partialapp', ['-contents',
 '/home/myProfile/apps/app1/app1Partial.zip'])
```

  where:

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
partialapp	is the content type value
contents	is an option of the **update** command
/apps/app1/app1Partial.zip	is the value of the contents option

- Update the entire deployed application.

  To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.

  - Using Jacl:

```
$AdminApp update app1 app {-operation update -contents
 /home/myProfile/apps/app1/newApp1.jar -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding {{"Increment EJB module"
 Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
```

  - Using Jython string:

```
AdminApp.update('app1', 'app', '[-operation update -contents
 /home/myProfile/apps/app1/ newApp1.ear -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment EJB module"
 Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

  - Using Jython list:

```
bindJndiForEJBValue = [["Increment EJB module", "Increment", "
 Increment.jar,META-INF/ejb-jar.xml", "Inc"]]  AdminApp.update('app1', 'app', ['-operation', 'update', '-contents',
 '/apps/app1/NewApp1.ear', '-usedefaultbindings', '-nodeployejb', `-BindJndiForEJBNonMessageBinding', bindJndiForEJBValue])
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application objects management
`update`	is an AdminApp command
`app1`	is the name of the application to update
`app`	is the content type value
`operation`	is an option of the **update** command
`update`	is the value of the operation option
`contents`	is an option of the **update** command
/apps/app1/newApp1.ear	is the value of the contents option
`usedefaultbindings`	is an option of the **update** command
`nodeployejb`	is an option of the **update** command
`BindJndiForEJBNonMessageBinding`	is an option of the **update** command

`"Increment EJB module" Increment Increment.jar,META-INF/ejb-jar.xml Inc`	is the value of the BindJndiForEJBNonMessageBinding option. The value of this option is defined in your application configuration. To determine the value of this option, use the following Jython or Jacl command:
	
	Using Jython:
	`AdminApp.view('myAppName')`
	
	Using Jacl:
	`$AdminApp view myAppName`

`bindJndiForEJBValue`	is a Jython variable containing the value of the BindJndiForEJBNonMessageBinding option

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     `AdminNodeManagement.syncActiveNodes()`
   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     `AdminNodeManagement.syncNode("myNode")`

What to do next

The steps in this task return a success message if the system successfully updates the application. When updating large applications, the command might return a success message before the system extracts each binary file. You cannot start the application until the system extracts all binary files. If you installed a large application, use the **isAppReady** and **getDeployStatus** commands for the AdminApp object to verify that the system extracted the binary files before starting the application.

The **isAppReady** command returns a value of `true` if the system is ready to start the application, or a value of `false` if the system is not ready to start the application.

```
AdminApp.isAppReady('myapp1')
```

If the system is not ready to start the application, the system might be expanding application binaries. Use the **getDeployStatus** command to display additional information about the binary file expansion status, as the following example displays:

```
AdminApp.getDeployStatus('app1')
```

Managing assets with scripting

Use the commands in the BLAManagement command group to manage your asset configuration. Use the examples in this topic to list assets, view asset configuration data, remove assets from the asset repository, update one or more files for assets, and export assets.

Before you begin

There are two ways to complete this task. Complete the tasks in this topic to manage assets with the BLAManagement command group for the AdminTask object. Alternatively, you can use the scripts in the AdminBLA script library to administer your asset configurations.

- List assets.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List the assets registered to the asset repository.

     Use the listAssets command to display the configuration ID, description, and deployment target for each asset within the cell, as the following command demonstrates:

```
AdminTask.listAssets()
```

- View asset settings.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Display the asset settings.

     Use the viewAsset command to display the configuration information for the asset of interest, as the following example demonstrates:

```
AdminTask.viewAsset('-assetID myAsset.zip')
```

     The command returns the configured asset options, as the following sample output displays:

```
Specify Asset options (AssetOptions)  Specify options for Asset.  *Asset Name (name):
 [defaultapp.ear] Default Binding Properties (defaultBindingProps):
 [defaultbinding.ejbjndi.prefix#defaultbinding.datasource.jndi#
  defaultbinding.datasource.username# defaultbinding.datasource.password# defaultbinding.cf.jndi#
  defaultbinding.cf.resauth#defaultbinding.virtual.host# defaultbinding.force]
  Asset Description (description): [] Asset Binaries
  Destination Url (destination): [${USER_INSTALL_ROOT}/installedAssets/defaultapp.ear/BASE/defaultapp.ear]
  Asset Type Aspects(typeAspect): [WebSphere:spec=j2ee_ear] Asset Relationships (relationship):
  []File Permission (filePermission):
  [.*\\.dll=755#.*\\.so=755#.*\\.a=755#.*\\.sl=755] Validate asset (validate): [false]
```

- Remove one or more assets from the product management domain.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine if the asset can be deleted.

     You cannot delete an asset from the asset registry if it is associated with composition unit in a business-level application. Use the listCompUnits command to display the configuration ID, type, and description for each composition unit in a business-level application, as the following example demonstrates:

```
AdminTask.listCompUnits('-blaID myBLA -includeDescription true')
```

     The command returns the following sample output:

```
Websphere:cuname=cu1 asset "Composition unit for asset.zip"  Websphere:cuname=cu4 bla  "cu4
 description" WebSphere:cuname=defaultapp __j2ee "defaultapp description"
```

The type for the `cu1` composition unit is `asset`, which denotes that the composition unit is associated with an asset. Use the deleteCompUnit command to remove the composition unit before deleting the asset from the asset repository, as the following example demonstrates:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1')
```

3. Delete the asset.

   Use the deleteAsset command to remove the asset of interest from the asset repository, as the following example demonstrates:

```
AdminTask.deleteAsset('-assetID asset2.zip')
```

   The command returns the configuration ID of the deleted asset, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

- Update the contents of an asset.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine how to update the asset.

     You can invoke several different operations on assets that are registered in the asset repository, as the following table displays:

Operation	Description
`replace`	The `replace` operation replaces the contents of the asset of interest.
`merge`	The `merge` operation updates multiple files for the asset, but does not update all files.
`add`	The `add` operation adds a new file or module file.
`addupdate`	The `addupdate` operation adds or updates one file or module file. If the file does not exist, the system adds the contents. If the file exists, the system updates the file.
`update`	The `update` operation updates one file or module file.
`delete`	The `delete` operation deletes a file or module file.

  3. Update the asset of interest.

     The updateAsset command modifies one or more files or module files of an asset, as the following example demonstrates:

```
AdminTask.updateAsset('-assetID asset2.zip –operation merge –contents
 \temp\updatedFiles_asset1.zip')
```

     The command updates the asset binary file, but does not update the composition unit that the system deploys with the asset as a backing object.
  4. Save your configuration changes.
- Export an asset to a target location.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Export the asset of interest.

     Use the exportAsset command to save an asset configuration to a file. The command accepts an incomplete asset configuration ID if the system matches it to a unique ID in your configuration. The following example exports an asset:

```
AdminTask.exportAsset('-assetID asset2.zip –filename \temp\a2.zip')
```

Managing composition units with scripting

Use the commands in the BLAManagement command group to manage composition units. Use the examples in this topic to add, remove, edit, export, and view composition units.

Before you begin

There are two ways to complete the examples in this task. Use the BLAManagement command group for the AdminTask object to manage composition units. Alternatively, you can use the scripts in the AdminBLA script library to administer your composition unit configurations.

About this task

Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-WebSphere Application Server runtime environments without associated assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

- Add composition units.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Add composition units.

     Use the addCompUnit command to add composition units to business-level applications. Use the following command example to add the `asset1` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` server:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset1 -CUOptions [[.* .*
 compositionUnit1 "composition unit that is backed by asset1" 0]] -MapTargets [[.* server1]]
 –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

     Use the following command to add the `asset2` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset2 -CUOptions [[.* .*
 compositionUnit2 "composition unit that is backed by asset2" 0]] -MapTargets [[.*
 server1+testServer]] –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

     Use the following command to add the `J2EEAsset` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('[-blaID myBLA –cuSourceID J2EEAsset
 –defaultBindingOptions defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host#
 defaultbinding.force=yes –AppDeploymentOptions [-appname defaultapp -installed.ear.destination
 application_root/myCell/defaultapp.ear] –MapModulesToServers [[defaultapp.war .* WebSphere:cell=cellName,node=nodeName,server=server1][Increment.jar .*
 Websphere:cell=cellName,node=nodeName,server=testServer]] -CtxRootForWebMod [[defaultapp.war .*
 myctx/]]]')
```

     The command returns the configuration IDs of the composition unit and the new composition unit created for the asset in the asset relationship, as the following example displays:

```
WebSphere:cuname=compositionUnit1 WebSphere:cuname=compositionUnit2
 WebSphere:cuname=J2EEAsset
```

  3. Save your configuration changes.
- Display composition units and configuration settings.

  Use the listCompUnits and viewCompUnits commands to display the configuration IDs of each composition unit that matches a specific search scope.

  You can use the listCompUnits command to display each composition unit in your configuration or within a specific business-level application. The following example displays each composition unit in the `myBLA` business-level application:

```
AdminTask.listCompUnits('-blaID blaname=myBLA')
```

  The command returns the configuration IDs and type of backing asset for each composition unit that matches the search scope, as the following sample displays:

```
Websphere:cuname=cu1 asset  Websphere:cuname=cu4 bla  WebSphere:cuname=defaultapp
 __j2ee
```

  You can use the viewCompUnits command to display additional configuration information about a specific composition unit of a business-level application. For example, the following example displays additional information about the `cu1` composition unit for the `myBLA` business-level application:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID cu1')
```

  The command returns detailed configuration information for the composition unit, as the following sample displays:

```
Specify Composition Unit options (CUOptions)  Specify name, description options for
 Composition Unit.  Parent BLA (parentBLA): [WebSphere:blaname=myBLA] Backing Id (backingId): [WebSphere:assetname=asset1.zip]
 Name (name): [cu1] Description (description): [my description of cu1 composition unit] Starting Weight (startingWeight): [0]
 Specify servers (MapTargets)  Specify targets such as application servers or clusters of application servers where you want to
 deploy the composition unit contained in the application.  Deployable Unit (deplUnit): [default] *Servers (server):
 [WebSphere:node=myNode,server=server1]  Specify Composition Unit activation plan options (ActivationPlanOptions)  Specify
 composition unit activation plan optionsDeployableUnit Name (deplUnit): [default] Activation Plan (activationPlan):
 [WebSphere:specname=actplan0+WebSphere:specname=actplan1]
```

Use the editCompUnit command to modify the composition unit configuration options.

- Edit composition units.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Modify the composition unit.

     Use the editCompUnit command to modify composition unit options. You can use this command to modify the starting weight of the composition unit, deployment targets, activation plan options, and relationship settings. See the documentation for the BLAManagement command group for the AdminTask object to view descriptions of each option that you can modify.

     The following example edits a composition unit, which is associated with an asset, and replaces the deployment target:

```
AdminTask.editCompUnit('-blaID myBLA –cuID cu1 –CUOptions [[.* .* cu1
 cudesc 1]] -MapTargets [[.* server2]] -ActivationPlanOptions [.*
 #specname=actplan0+specname=actplan2]')
```

     The command returns the configuration ID of the composition unit that the system edits, as the following sample displays:

```
WebSphere:cuname=cu1
```

  3. Save your configuration changes.
- Remove composition units.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Remove composition units.

     Use the deleteCompUnit command to remove a composition unit. Both parameters for the following command accept incomplete configuration IDs, as long as the system can match the string to a unique ID:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1')
```

     The command returns the configuration ID of the composition unit that the system deletes, as the following sample demonstrates:

```
WebSphere:cuname=cu1
```

  3. Save your configuration changes.

Listing the modules in an installed application with scripting

Use the AdminApp object **listModules** command to list the modules in an installed application.

Before you begin

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Display the application modules.

   Using Jacl:

```
$AdminApp listModules DefaultApplication -server
```

   Using Jython:

```
print AdminApp.listModules('DefaultApplication', '-server')
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`print`	is a Jython command
`AdminApp`	is an object that supports application object management
`listmodules`	is an AdminApp command
`DefaultApplication`	is the name of the application
`-server`	is an optional option specified

Example output:

```
DefaultApplication#IncCMP11.jar+META-INF/ejb-jar.xml#WebSphere:cell=mycell,node=mynode,server=myserver
DefaultApplication#DefaultWebApplication.war+WEB-INF/web.xml#WebSphere:cell=mycell,node=mynode,server=myserver
```

Example: Listing the modules in an application server

This example lists all of the modules on all of the enterprise applications that are installed on the `server1` server in a node named `node1`.

An asterisk (*****) means that the module is installed on server1 and node1 and another server or node. A plus sign (**+**) means that the module is installed on server1 and node1 only.

```
 1  #-------------------------------------------------------------------------------
       2  # setting up variables to keep server name and node name
       3  #-------------------------------------------------------------------------------
       4  set  serverName  server1
       5  set  nodeName  node1
       6  #-------------------------------------------------------------------------------
       7  # setting up 2 global lists to keep the modules
       8  #-------------------------------------------------------------------------------
       9  set  ejbList {}
      10 set webList {}
11
12 #-------------------------------------------------------------------------------------------------
13 # gets all deployment objects and assigned it to deployments variable
14 #-------------------------------------------------------------------------------------------------
15 set deployments [$AdminConfig getid /Deployment:/]
16
17 #-------------------------------------------------------------------------------------------------
18 # lines 22 thru 148 Iterates through all the deployment objects to get the modules
19 # and perform filtering to list application that has at least one module installed
20 # in server1 in node myNode
21 #-------------------------------------------------------------------------------------------------
22 foreach deployment $deployments {
23
24     # ---------------------------------------------------------------------------------------
25     # reset the lists that hold modules for each application
26     #----------------------------------------------------------------------------------------
27     set webList {}
28     set ejbList {}
29
30     #-----------------------------------------
31     # get the application name
32     #-----------------------------------------
33     set appName [lindex [split $deployment (] 0]
34
35     #-----------------------------------------
36     # get the deployedObjects
37     #-----------------------------------------
38     set depObject [$AdminConfig showAttribute $deployment deployedObject]
39
40     #-----------------------------------------
41     # get all modules in the application
42     #-----------------------------------------
43     set modules [lindex [$AdminConfig showAttribute $depObject modules] 0]
44
45     #-----------------------------------------------------------------------------------------------
46     # initialize lists to save all the modules in the appropriate list to where they belong
47     #-----------------------------------------------------------------------------------------------
48     set modServerMatch {}
49     set modServerMoreMatch {}
50     set modServerNotMatch {}
51
52         #-------------------------------------------------------------------------------
53         # lines 55 to 112 iterate through all modules to get the targetMappings
54         #-------------------------------------------------------------------------------
```

```
55          foreach module $modules {
56              #-------------------------------------------------------------------------------------------------
57              # setting up some flag to do some filtering and get modules for server1 on node1
58              #-------------------------------------------------------------------------------------------------
59              set sameNodeSameServer "false"
60             set diffNodeSameServer "false"
61              set sameNodeDiffServer "false"
62              set diffNodeDiffServer "false"
63
64              #-----------------------------------------
65              # get the targetMappings
66              #-----------------------------------------
67              set targetMaps [lindex [$AdminConfig showAttribute $module targetMappings] 0]
68
69              #-------------------------------------------------------------------------------------------------
70             # lines 72 to 111 iterate through all targetMappings to get the target
71              #-------------------------------------------------------------------------------------------------
72              foreach targetMap $targetMaps {
73                  #------------------------------
74                 # get the target
75                  #------------------------------
76                  set target [$AdminConfig showAttribute $targetMap target]
77
78                  #-------------------------------------------------
79                  # do filtering to skip ClusteredTargets
80                  #-------------------------------------------------
81                  set targetName [lindex [split $target #] 1]
82                 if {[regexp "ClusteredTarget" $targetName] != 1} {
83                      set sName [$AdminConfig showAttribute $target name]
84                      set nName [$AdminConfig showAttribute $target nodeName]
85
86                      #-----------------------------------------
87                      # do the server name match
88                      #-----------------------------------------
89                      if {$sName == $serverName} {
90                        if {$nName == $nodeName} {
91                  set sameNodeSameServer "true"
92                          } else {
93                  set diffNodeSameServer "true"
94                          }
95                     } else {
96                           #---------------------------------------
97                           # do the node name match
98                           #---------------------------------------
99                           if {$nName == $nodeName} {
100                  set sameNodeDiffServer "true"
101                        } else {
102                            set diffNodeDiffServer "true"
103                        }
104                  }
105
106                  if {$sameNodeSameServer == "true"} {
107                      if {$sameNodeDiffServer == "true" || $diffNodeDiffServer == "true" ||
          $diffNodeSameServer == "true"} {
108                           break
109                       }
110                  }
111             }
112     }
113
114     #-----------------------------------------
115     # put it in the appropriate list
116     #-----------------------------------------
117     if {$sameNodeSameServer == "true"} {
118         if {$diffNodeDiffServer == "true" || $diffNodeSameServer == "true" || $sameNodeDiffServer == "true"} {
119             set modServerMoreMatch [linsert $modServerMoreMatch end [$AdminConfig showAttribute $module uri]]
120          } else {
```

```
121              set modServerMatch [linsert $modServerMatch end [$AdminConfig showAttribute $module uri]]
122          }
123     } else {
124          set modServerNotMatch [linsert $modServerNotMatch end [$AdminConfig showAttribute $module uri]]
125     }
126 }
127
128
129 #-------------------------------------------------------------
130 # print the output with some notation as a mark
131 #-------------------------------------------------------------
132 if {$modServerMatch != {} || $modServerMoreMatch != {}} {
133     puts stdout "\tApplication name: $appName"
 134 }
 135
 136 #---------------------------------------------------------
 137 # do grouping to appropriate module and print
 138 #---------------------------------------------------------
 139 if {$modServerMatch != {}} {
 140     filterAndPrint $modServerMatch "+"
 141 }
 142 if {$modServerMoreMatch != {}} {
 143     filterAndPrint $modServerMoreMatch "*"
 144 }
 145 if {($modServerMatch != {} || $modServerMoreMatch != {}) "" $modServerNotMatch != {}} {
 146     filterAndPrint $modServerNotMatch ""
 147 }
 148}
 149
 150
 151 proc filterAndPrint {lists flag} {
 152    global webList
 153    global ejbList
 154    set webExists "false"
 155    set ejbExists "false"
 156
 157    #-------------------------------------------------------------------------------------------------------
 158    # If list already exists, flag it so as not to print the title more then once
 159    # and reset the list
 160    #-------------------------------------------------------------------------------------------------------
 161    if {$webList != {}} {
 162        set webExists "true"
 163        set webList {}
 164    }
 165    if {$ejbList != {}} {
 166        set ejbExists "true"
 167        set ejbList {}
 168    }
 169
 170    #-----------------------------------------------------------------
 171    # do some filtering for web modules and ejb modules
 172    #-----------------------------------------------------------------
 173    foreach list $lists {
 174         set temp [lindex [split $list .] 1]
 175         if {$temp == "war"} {
 176              set webList [linsert $webList end $list]
 177         } elseif {$temp == "jar"} {
 178              set ejbList [linsert $ejbList end $list]
 179         }
 180    }
 181
 182    #---------------------------------------
 183    # sort the list before printing
 184    #---------------------------------------
 185    set webList [lsort -dictionary $webList]
 186    set ejbList [lsort -dictionary $ejbList]
 187
```

```
188     #-------------------------------------------------------------
189     # print out all the web modules installed in server1
190     #-------------------------------------------------------------
191     if {$webList != {}} {
192         if {$webExists == "false"} {
193              puts stdout "\t\tWeb Modules:"
194         }
195         foreach web $webList {
196             puts stdout "\t\t\t$web  $flag"
197         }
198     }
199
200     #-------------------------------------------------------------
201     # print out all the ejb modules installed in server1
202     #-------------------------------------------------------------
203     if {$ejbList != {}} {
204          if {$ejbExists == "false"} {
205               puts stdout "\t\tEJB Modules:"
206          }
207          foreach ejb $ejbList {
208               puts stdout "\t\t\t$ejb  $flag"
209          }
210     }
211}
```

```
Example output for server1 on node node1:
       Application name: TEST1
                EJB Modules:
                        deplmtest.jar  +
                Web Modules:
                        mtcomps.war  *
        Application name: TEST2
                Web Modules:
                        mtcomps.war  +
                EJB Modules:
                        deplmtest.jar  +
        Application name: TEST3
                Web Modules:
                        mtcomps.war  *
                EJB Modules:
                        deplmtest.jar  *
        Application name: TEST4
                EJB Modules:
                        deplmtest.jar  *
                Web Modules:
                        mtcomps.war
```

Querying the application state using scripting

Use the wsadmin tool and scripting to determine if an application is running.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the application state.

The following example queries the presence of the Application MBean to find out whether the application is running.

- Using Jacl:

```
$AdminControl completeObjectName type=Application,name=myApplication,*
```

- Using Jython:

```
print AdminControl.completeObjectName('type=Application,name=myApplication,*')
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
`type=Application,name=`*`myApplication`*	is the hierarchical containment path of the configuration object
`print`	is a Jython command

Results

If *myApplication* is running, then an MBean is created. Otherwise, the command returns nothing. If *myApplication* is running, the output would resemble the following:

```
WebSphere:cell=mycell,name=myApplication,mbeanIdentifier=cells/mycell/applications/myApplication.ear/
deployments/myApplication/deployment.xml#ApplicationDeployment_1,type=Application,node=mynode,Server=
dmgr,process=dmgr,J2EEName=myApplication
```

Disabling application loading in deployed targets using scripting

You can use the AdminConfig object and scripting to disable application loading in deployed targets.

About this task

The following example uses the AdminConfig object to disable application loading in deployed targets:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Obtain the Deployment object for the application and assign it to the deployments variable, for example:
   - Using Jacl:

     ```
     set deployments [$AdminConfig getid /Deployment:myApp/]
     ```

   - Using Jython:

     ```
     deployments = AdminConfig.getid("/Deployment:myApp/")
     ```

   where:

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
`myApp`	is the value of the attribute

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

3. Obtain the target mappings in the application and assign them to the targetMappings variable, for example:
   - Using Jacl:

     ```
     set deploymentObj1 [$AdminConfig showAttribute $deployments deployedObject]
     set targetMap1 [lindex [$AdminConfig showAttribute $deploymentObj1 targetMappings] 0]
     ```

     Example output:

     ```
     (cells/mycell/applications/ivtApp.ear/deployments/ivtApp|deployment.xml#DeploymentTargetMapping_1)
     ```

   - Using Jython:

     ```
     deploymentObj1 = AdminConfig.showAttribute(deployments, 'deployedObject')
     targetMap1 = AdminConfig.showAttribute(deploymentObj1, 'targetMappings')
     targetMap1 = targetMap1[1:len(targetMap1)-1].split(" ")
     print targetMap1
     ```

     Example output:

     ```
     ['(cells/mycell/applications/ivtApp.ear/deployments/ivtApp|deployment.xml#DeploymentTargetMapping_1)']
     ```

   where:

`set`	is a Jacl command
`deploymentObj1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates the ID of the Deployment object that is specified in step number 1
`deployedObject`	is an attribute
`targetMap1`	is a variable name
`targetMappings`	is an attribute
`lindex`	is a Jacl command
`print`	is a Jython command

4. Disable the loading of the application on each deployed target, for example:
   - Using Jacl:

     ```
     foreach tm $targetMap1 {
           $AdminConfig modify $tm {{enable false}}
     }
     ```

   - Using Jython:

     ```
     for targetMapping in targetMap1:
        AdminConfig.modify(targetMapping, [["enable", "false"]])
     ```

5. Save the configuration changes.
6. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

```
AdminNodeManagement.syncNode("myNode")
```

Configuring applications for session management using scripting

This task provides an example that uses the AdminConfig object to configure a session manager for the application.

Before you begin

About this task

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the deployment configuration object for the application and assign it to the deployment variable. For example:
   - Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:myApp/]
```

   - Using Jython:

```
deployments = AdminConfig.getid('/Deployment:myApp/')
print deployments
```

   where:

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
myApp	is the value of the attribute

   Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

3. Retrieve the application deployment object and assign it to the appDeploy variable. For example:
   - Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployments deployedObject]
```

   - Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployments, 'deployedObject')
print appDeploy
```

   where:

`set`	is a Jacl command
`appDeploy`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command

deployments	evaluates the ID of the deployment object that is specified in step number 1
deployedObject	is an attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#ApplicationDeployment_1)
```

4. To obtain a list of attributes that you can set for a session manager, use the **attributes** command. For example:
   - Using Jacl:

```
$AdminConfig attributes SessionManager
```

   - Using Jython:

```
print AdminConfig.attributes('SessionManager')
```

   where:

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
attributes	is an AdminConfig command
SessionManager	is an attribute

Example output:

```
"accessSessionOnTimeout Boolean"
"allowSerializedSessionAccess Boolean"
"context ServiceContext@"
"defaultCookieSettings Cookie"
"enable Boolean"
"enableCookies Boolean"
"enableProtocolSwitchRewriting Boolean"
"enableSSLTracking Boolean"
"enableSecurityIntegration Boolean"
"enableUrlRewriting Boolean"
"maxWaitTime Integer"
"properties Property(TypedProperty)*"
"sessionDRSPersistence DRSSettings"
"sessionDatabasePersistence SessionDatabasePersistence"
"sessionPersistenceMode ENUM(DATABASE, DATA_REPLICATION, NONE)"
"tuningParams TuningParams"
```

When you configure and application for session management, it is recommended that you specify each attribute.

5. Set up the attributes for the session manager. The following example sets four top-level attributes in the session manager. You can modify the example to set other attributes of the session manager, including the nested attributes in DRSSettings, SessionDataPersistence, and TuningParms object types. To list the attributes for those object types, use the **attributes** command of the AdminConfig object.
   - Using Jacl:

```
set attr1 [list enableSecurityIntegration true]
set attr2 [list maxWaitTime 30]
set attr3 [list sessionPersistenceMode NONE]
set kuki [list maximumAge -1]
set cookie [list $kuki]
Set cookieSettings [list defaultCookieSettings $cookie]
set attrs [list $attr1 $attr2 $attr3 $cookieSettings]
set sessionMgr [list sessionManagement $attrs]
```

Example output using Jacl:

```
sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30} {sessionPersistenceMode NONE}
{defaultCookieSettings {{maximumAge -1}}}}
```

   - Using Jython:

```
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
kuki = ['maximumAge', -1]
cookie = [kuki]
cookieSettings = ['defaultCookieSettings', cookie]
attrs = [attr1, attr2, attr3, cookieSettings]
sessionMgr = [['sessionManagement', attrs]]
```

Example output using Jython:

```
[[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30], [sessionPersistenceMode, NONE],
[defaultCookieSettings [[maximumAge, -1]]]]
```

where:

set	is a Jacl command
attr1, attr2, attr3, attrs, sessionMgr	are variable names
$	is a Jacl operator for substituting a variable name with its value
enableSecurityIntegration	is an attribute
true	is a value of the enableSecurityIntegration attribute
maxWaitTime	is an attribute
30	is a value of the maxWaitTime attribute
sessionPersistenceMode	is an attribute
NONE	is a value of the sessionPersistenceMode attribute

6. Perform one of the following:
   - Create the session manager for the application. For example:
     - Using Jacl:

```
$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr]
```

     - Using Jython:

```
print AdminConfig.create('ApplicationConfig', appDeploy, sessionMgr)
```

     where:

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
create	is an AdminConfig command
ApplicationConfig	is an attribute
appDeploy	evaluates the ID of the deployed application that is specified in step number 2
list	is a Jacl command
sessionMgr	evaluates the ID of the session manager that is specified in step number 4

     Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#ApplicationConfig_1)
```

   - If a session manager already exists, use the **modify** command of the AdminConfig object to update the configuration of the session manager. For example:
     - Using Jacl:

```
set configs [lindex [$AdminConfig showAttribute $appDeploy configs] 0]
set appConfig [lindex $configs 0]
set SM [$AdminConfig showAttribute $appConfig sessionManagement]
$AdminConfig modify $SM $attrs
```

– Using Jython:

```
configs = AdminConfig.showAttribute (appDeploy, 'configs')
appConfig = configs[1:len(configs)-1]
SM = AdminConfig.showAttribute (appConfig, 'sessionManagement')
AdminConfig.modify (SM, attrs)
```

7. Save the configuration changes.
8. Synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Related concepts

Session management support
WebSphere Application Server provides facilities, grouped under the heading *Session Management*, that support the javax.servlet.http.HttpSession interface described in the Servlet API specification.

Related tasks

"Configuring applications for session management in Web modules using scripting"
Use scripting and the wsadmin tool to configure applications for session management in Web modules.

Configuring session management by level
When you configure session management at the Web container level, all applications and the respective Web modules in the Web container normally inherit that configuration, setting up a basic default configuration for the applications and Web modules below it. However, you can set up different configurations individually for specific applications and Web modules that vary from the Web container default. These different configurations override the default for these applications and Web modules only.

Developing session management in servlets

Chapter 3, "Using the script library to automate the application serving environment," on page 81
The script library provides Jython script procedures to assist in automating your environment. Use the sample scripts to manage applications, resources, servers, nodes, and clusters. You can also use the script procedures as examples to learn the Jython syntax.

"Using the AdminConfig object for scripted administration" on page 41
Use the AdminConfig object to manage the configuration information that is stored in the repository.

Configuring applications for session management in Web modules using scripting

Use scripting and the wsadmin tool to configure applications for session management in Web modules.

Before you begin

About this task

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object. The following task uses the AdminConfig object to configure a session manager for a Web module in the application.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the deployment configuration object for the application and assign it to the deployment variable. For example:
   - Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:myApp/]
```

- Using Jython:

```
deployments = AdminConfig.getid('/Deployment:myApp/')
print deployments
```

where:

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
myApp	is the value of the attribute

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

3. Get all the modules in the application and assign them to the modules variable. For example:

- Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployments deployedObject]
set mod1 [$AdminConfig showAttribute $appDeploy modules]
```

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#WebModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#EJBModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#WebModuleDeployment_2)
```

- Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployments, 'deployedObject')
mod1 = AdminConfig.showAttribute(appDeploy, 'modules')
print mod1
```

Example output:

```
[(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#WebModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#EJBModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#EJBModuleDeployment_2)]
```

where:

`set`	is a Jacl command
`appDeploy`	is a variable name
`mod1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates the ID of the deployment object that is specified in step number 1
`deployedObject`	is an attribute

4. To obtain a list of attributes that you can set for a session manager, use the **attributes** command. For example:

- Using Jacl:

```
$AdminConfig attributes SessionManager
```

- Using Jython:

```
print AdminConfig.attributes('SessionManager')
```

where:

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
attributes	is an AdminConfig command
SessionManager	is an attribute

Example output:

```
"accessSessionOnTimeout Boolean"
"allowSerializedSessionAccess Boolean"
"context ServiceContext@"
"defaultCookieSettings Cookie"
"enable Boolean"
"enableCookies Boolean"
"enableProtocolSwitchRewriting Boolean"
"enableSSLTracking Boolean"
"enableSecurityIntegration Boolean"
"enableUrlRewriting Boolean"
"maxWaitTime Integer"
"properties Property(TypedProperty)*"
"sessionDRSPersistence DRSSettings"
"sessionDatabasePersistence SessionDatabasePersistence"
"sessionPersistenceMode ENUM(DATABASE, DATA_REPLICATION, NONE)"
"tuningParams TuningParams"
```

5. Set up the attributes for session manager. The following example sets four top-level attributes in the session manager. You can modify the example to set other attributes in the session manager, including the nested attributes in Cookie, DRSSettings, SessionDataPersistence, and TuningParms object types. To list the attributes for those object types, use the **attributes** command of AdminConfig object.

   - Using Jacl:

```
set attr0 [list enable true]
set attr1 [list enableSecurityIntegration true]
set attr2 [list maxWaitTime 30]
set attr3 [list sessionPersistenceMode NONE]
set attr4 [list enableCookies true]
set attr5 [list invalidationTimeout 45]
set tuningParmsDetailList [list $attr5]
set tuningParamsList [list tuningParams $tuningParmsDetailList]
set pwdList [list password 95ee608]
set userList [list userId Administrator]
set dsNameList [list datasourceJNDIName jdbc/session]
set dbPersistenceList [list $dsNameList $userList $pwdList]
set sessionDBPersistenceList [list $dbPersistenceList]
set sessionDBPersistenceList [list sessionDatabasePersistence $dbPersistenceList]
set kuki [list maximumAge 1000]
set cookie [list $kuki]
set cookieSettings [list defaultCookieSettings $cookie]
set sessionManagerDetailList [list $attr0 $attr1 $attr2 $attr3 $attr4 $cookieSettings
$tuningParamsList $sessionDBPersistenceList]
set sessionMgr [list sessionManagement $sessionManagerDetailList]
set id [$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr] configs]
set targetMappings [lindex [$AdminConfig showAttribute $appDeploy targetMappings] 0]
set attrs [list config $id]
$AdminConfig modify $targetMappings [list $attrs]
```

   Example output using Jacl:

```
sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30}
{sessionPersistenceMode NONE} {enabled true}}
```

   - Using Jython:

```
attr0 = ['enable', 'true']
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
attr4 = ['enableCookies', 'true']
```

```
attr5 = ['invalidationTimeout', 45]
tuningParmsDetailList = [attr5]
tuningParamsList = ['tuningParams', tuningParmsDetailList]
pwdList = ['password', '95ee608']
userList = ['userId', 'Administrator']
dsNameList = ['datasourceJNDIName', 'jdbc/session']
dbPersistenceList = [dsNameList, userList, pwdList]
sessionDBPersistenceList = [dbPersistenceList]
sessionDBPersistenceList = ['sessionDatabasePersistence', dbPersistenceList]
kuki = ['maximumAge', 1000]
cookie = [kuki]
cookieSettings = ['defaultCookieSettings', cookie]
sessionManagerDetailList = [attr0, attr1, attr2, attr3, attr4, cookieSettings,
tuningParamsList, sessionDBPersistenceList]
sessionMgr = ['sessionManagement', sessionManagerDetailList]
id = AdminConfig.create('ApplicationConfig', appDeploy,[sessionMgr], 'configs')
targetMappings = AdminConfig.showAttribute(appDeploy, 'targetMappings')
targetMappings = targetMappings[1:len(targetMappings)-1]
print targetMappings
attrs = ['config', id]
AdminConfig.modify(targetMappings,[attrs])
```

Example output using Jython:

```
[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30], [sessionPersistenceMode, NONE]]
```

6. Set up the attributes for the Web module. For example:
   - Using Jacl:

```
set nameAttr [list name myWebModuleConfig]
set descAttr [list description "Web Module config post create"]
set webAttrs [list $nameAttr $descAttr $sessionMgr]
```

Example output:

```
{name myWebModuleConfig} {description {Web Module config post create}}
{sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30}
{sessionPersistenceMode NONE} {enabled true}}}
```

   - Using Jython:

```
nameAttr = ['name', 'myWebModuleConfig']
descAttr = ['description', "Web Module config post create"]
webAttrs = [nameAttr, descAttr, sessionMgr]
```

Example output:

```
[[name, myWebModuleConfig], [description, "Web Module config post create"],
[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30],
[sessionPersistenceMode, NONE], [enabled, true]]]]
```

where:

`set`	is a Jacl command
`nameAttr`, `descAttr`, `webAttrs`	are variable names
`$`	is a Jacl operator for substituting a variable name with its value
`name`	is an attribute
myWebModuleConfig	is a value of the name attribute
`description`	is an attribute
Web Module config post create	is a value of the description attribute

7. Create the session manager for each Web module in the application. You can modify the following example to set other attributes of the session manager in a Web module configuration. You must also define a target mapping for this step.
   - Using Jacl:

```
foreach module $mod1 {
 if {[regexp WebModuleDeployment $module] == 1} {
  set moduleConfig [$AdminConfig create WebModuleConfig $module $webAttrs]
  set targetMappings [lindex [$AdminConfig showAttribute $module targetMappings] 0]
```

```
  set attrs [list config $moduleConfig]
  $AdminConfig modify $targetMappings [list $attrs]
 }
}
```

- Using Jython:

```
arrayModules = mod1[1:len(mod1)-1].split(" ")
for module in arrayModules:
 if module.find('WebModuleDeployment') != -1:
  AdminConfig.create('WebModuleConfig', module, webAttrs)
  targetMappings = targetMappings[1:len(targetMappings)-1]
  attrs = ['config', moduleConfig]
  AdminConfig.modify (targetMappings, [attrs])
```

Example output:

```
myWebModuleConfig(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#WebModuleConfiguration_1)
```

If you do not specify the tuningParamsList attribute when you create the session manager, you will receive an error when you start the deployed application.

8. Save the configuration changes.
9. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Related concepts

Session management support
WebSphere Application Server provides facilities, grouped under the heading *Session Management*, that support the javax.servlet.http.HttpSession interface described in the Servlet API specification.

Related tasks

“Configuring applications for session management using scripting” on page 268
This task provides an example that uses the AdminConfig object to configure a session manager for the application.

Configuring session management by level
When you configure session management at the Web container level, all applications and the respective Web modules in the Web container normally inherit that configuration, setting up a basic default configuration for the applications and Web modules below it. However, you can set up different configurations individually for specific applications and Web modules that vary from the Web container default. These different configurations override the default for these applications and Web modules only.

Developing session management in servlets

Chapter 3, “Using the script library to automate the application serving environment,” on page 81
The script library provides Jython script procedures to assist in automating your environment. Use the sample scripts to manage applications, resources, servers, nodes, and clusters. You can also use the script procedures as examples to learn the Jython syntax.

“Using the AdminConfig object for scripted administration” on page 41
Use the AdminConfig object to manage the configuration information that is stored in the repository.

Exporting applications using scripting

You can export your applications before you update installed applications or before you migrate to a different version of the WebSphere Application Server product.

Before you begin

About this task

Exporting applications enables you to back them up and preserve their binding information.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Export applications.
   - Export an enterprise application to a location of your choice, for example:
     – Using Jacl:
     – Using Jython:

       where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects management
`export`	is an AdminApp command
app1	is the name of the application that will be exported
`/mystuff/exported.ear`	is the name of the file where the exported application will be stored

   - Export Data Definition Language (DDL) files in the enterprise bean module of an application to a destination directory, for example:
     – Using Jacl:
     – Using Jython:

     where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects management
`exportDDL`	is an AdminApp command
app1	is the name of the application whose DDL files will be exported
/mystuff	is the name of the directory where the DDL files export from the application

Configuring a shared library using scripting

You can use scripting to configure a shared library for application servers. Shared libraries are files used by multiple applications. Create a shared library to reduce the number of duplicate library files on your system.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

1. Launch the wsadmin scripting tool using the Jython scripting language.

2. Identify the server and assign it to the server variable. For example:
   - Using Jacl:

```
set serv [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
serv = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print serv
```

   where:

`set`	is a Jacl command
`serv`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is an attribute
`mycell`	is the value of the attribute
`Node`	is an attribute
`mynode`	is the value of the attribute
`Server`	is an attribute
`server1`	is the value of the attribute

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Create the shared library in the server. For example:
   - Using Jacl:

```
$AdminConfig create Library $serv {{name mySharedLibrary} {classPath
/home/myProfile/mySharedLibraryClasspath}}
```

   - Using Jython:

```
print AdminConfig.create('Library', serv, [['name', 'mySharedLibrary'], ['classPath',
'home/myProfile/mySharedLibraryClasspath']])
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Library`	is an attribute
`serv`	evaluates the ID of the server that is specified in step number 1
`name`	is an attribute
`mySharedLibrary`	is a value of the name attribute
`classPath`	is an attribute
/mySharedLibraryClasspath	is the value of the classpath attribute
`print`	is a Jython command

   Example output:

```
MysharedLibrary(cells/mycell/nodes/mynode/servers/server1|libraries.xml#Library_1)
```

4. Identify the application server from the server and assign it to the appServer variable. For example:
   - Using Jacl:

```
set appServer [$AdminConfig list ApplicationServer $serv]
```

   - Using Jython:

```
appServer = AdminConfig.list('ApplicationServer', serv)
print appServer
```

   where:

set	is a Jacl command
appServer	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
list	is an AdminConfig command
ApplicationServer	is an attribute
serv	evaluates the ID of the server that is specified in step number 1
print	is a Jython command

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#ApplicationServer_1
```

5. Identify the class loader in the application server and assign it to the classLoader variable. For example:
   - To use the existing class loader that is associated with the server, the following commands use the first class loader:
     - Using Jacl:

```
set classLoad [$AdminConfig showAttribute $appServer classloaders]
set classLoader1 [lindex $classLoad 0]
```

     - Using Jython:

```
classLoad = AdminConfig.showAttribute(appServer, 'classloaders')
cleanClassLoaders = classLoad[1:len(classLoad)-1]
classLoader1 = cleanClassLoaders.split(' ')[0]
```

     where:

set	is a Jacl command
classLoad, classLoader1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
appServer	evaluates the ID of the application server that is specified in step number 3
classloaders	is an attribute
print	is a Jython command

   - To create a new class loader, issue the following command:
     - Using Jacl:

```
set classLoader1 [$AdminConfig create Classloader $appServer {{mode PARENT_FIRST}}]
```

– Using Jython:

```
classLoader1 = AdminConfig.create('Classloader', appServer, [['mode',  'PARENT_FIRST']])
```

where:

set	is a Jacl command
classLoader1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
create	is an AdminConfig command
Classloader	is an attribute
appServer	evaluates the ID of the application server that is specified in step number 3
mode	is an attribute
PARENT_FIRST	is the value of the attribute
print	is a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#Classloader_1)
```

6. Associate the shared library that you created with the application server through the class loader. For example:
   - Using Jacl:

```
$AdminConfig create LibraryRef $classLoader1 {{libraryName MyshareLibrary}}
```

   - Using Jython:

```
print AdminConfig.create('LibraryRef', classLoader1, [['libraryName', 'MyshareLibrary']])
```

   where:

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
create	is an AdminConfig command
LibraryRef	is an attribute
classLoader1	evaluates the ID of the class loader that is specified in step number 4
libraryName	is an attribute
MyshareLibrary	is the value of the attribute
print	is a Jython command

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#LibraryRef_1)
```

7. Save the configuration changes.
8. Synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

  ```
  AdminNodeManagement.syncNode("myNode")
  ```

Configuring a shared library for an application using scripting

This task uses the AdminConfig object to configure a shared library for an application. Shared libraries are files used by multiple applications. Create a shared library to reduce the number of duplicate library files on your system.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the shared library and assign it to the library variable. You can either use an existing shared library or create a new one, for example:
   - To create a new shared library, perform the following steps:
     a. Idenitfy the node and assign it to a variable, for example:
        – Using Jacl:

```
set n1 [$AdminConfig getid /Cell:mycell/Node:mynode/]
```

        – Using Jython:

```
n1 = AdminConfig.getid('/Cell:mycell/Node:mynode/')
print n1
```

        where:

`set`	is a Jacl command
`n1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is the object type
`mycell`	is the name of the object that will be modified
`Node`	is the object type
`mynode`	is the name of the object that will be modified

        Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

     b. Create the shared library in the node. The following example creates a new shared library in the node scope. You can modify it to use the cell or server scope.
        – Using Jacl:
        – Using Jython:

        where:

`set`	is a Jacl command

`library`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Library`	is an AdminConfig object
`n1`	evaluates to the ID of host node specified in step number 1
`name`	is an attribute
`mySharedLibrary`	is the value of the name attribute
`classPath`	is an attribute
/mySharedLibraryClasspath	is the value of the classPath attribute

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode|libraries.xml#Library_1)
```

- To use an existing shared library, issue the following command:
  - Using Jacl:

```
set library [$AdminConfig getid /Library:mySharedLibrary/]
```

  - Using Jython:

```
library = AdminConfig.getid('/Library:mySharedLibrary/')
print library
```

where:

`set`	is a Jacl command
`library`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Library`	is an attribute
`mySharedLibrary`	is the value of the Library attribute

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode|libraries.xml#Library_1)
```

3. Identify the deployment configuration object for the application and assign it to the deployment variable. For example:
   - Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:myApp/]
```

   - Using Jython:

```
deployment = AdminConfig.getid('/Deployment:myApp/')
print deployment
```

where:

`set`	is a Jacl command
`deployment`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value

AdminConfig	is an object representing the WebSphere Application Server configuration
getid	is an AdminConfig command
Deployment	is an attribute
myApp	is the value of the Deployment attribute
print	is a Jython command

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

4. Retrieve the application deployment and assign it to the appDeploy variable. For example:
   - Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployment deployedObject]
```

   - Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployment, 'deployedObject')
print appDeploy
```

   where:

set	is a Jacl command
appDeploy	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
deployment	evaluates the ID of the deployment configuration object specified in step number 2
deployedObject	is an attribute of modify objects
print	is a Jython command

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/
myApp|deployment.xml#ApplicationDeployment_1)
```

5. Identify the class loader in the application deployment and assign it to the classLoader variable. For example:
   - Using Jacl:

```
set classLoad1 [$AdminConfig showAttribute $appDeploy classloader]
```

   - Using Jython:

```
classLoad1 = AdminConfig.showAttribute(appDeploy, 'classloader')
print classLoad1
```

   where:

set	is a Jacl command
classLoad1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
appDeploy	evaluates the ID of the application deployment specified in step number 3

`classLoader`	is an attribute of modify objects
`print`	is a Jython command

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Classloader_1)
```

6. Associate the shared library in the application through the class loader. For example:
   - Using Jacl:

```
$AdminConfig create LibraryRef $classLoad1 {{libraryName
MyshareLibrary}}
```

   - Using Jython:

```
print AdminConfig.create('LibraryRef', classLoad1, [['libraryName',
'MyshareLibrary']])
```

   where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`create`	is an AdminConfig command
`LibraryRef`	is an AdminConfig object
`classLoad1`	evaluates to the ID of class loader specified in step number 4
`libraryName`	is an attribute
`MyshareLibrary`	is the value of the libraryName attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#LibraryRef_1)
```

7. Save the configuration changes.
8. Synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Setting background applications using scripting

You can enable or disable a background application using scripting and the wsadmin tool.

About this task

Background applications specify whether the application must initialize fully before the server starts. The default setting is `false` and this indicates that server startup will not complete until the application starts. If you set the value to `true`, the application starts on a background thread and server startup continues without waiting for the application to start. The application may not ready for use when the application server starts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Locate the application deployment object for the application. For example:

- Using Jacl:

  ```
  set applicationDeployment [$AdminConfig getid /Deployment:adminconsole/ApplicationDeployment:/]
  ```

- Using Jython:

  ```
  applicationDeployment = AdminConfig.getid('/Deployment:adminconsole/ApplicationDeployment:/')
  ```

  where:

`set`	is a Jacl command
`applicationDeployment`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is a type
`ApplicationDeployment`	is a type
`adminconsole`	is the name of the application

3. Enable the background application. For example:
   - Using Jacl:

     ```
     $AdminConfig modify $applicationDeployment "{backgroundApplication true}"
     ```

   - Using Jython:

     ```
     AdminConfig.modify(applicationDeployment, ['backgroundApplication', 'true'])
     ```

     where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`applicationDeployment`	is a variable name that was set in step 1
`backgroundApplication`	is an attribute
`true`	is the value of the backgroundApplication attribute

4. Save the configuration changes.
5. Synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Modifying WAR class loader policies for applications using scripting

You can use scripting and the wsadmin tool to modify WAR class loader policies for applications.

Before you begin

About this task

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

To modify WAR class loader policies for an application, perform the following steps:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:

     ```
     set dep [$AdminConfig getid /Deployment:MyApp/]
     ```
   - Using Jython:

     ```
     dep = AdminConfig.getid("/Deployment:MyApp/")
     ```
3. Identify the deployed object and set it to the deployedObject variable. For example:
   - Using Jacl:

     ```
     set deployedObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:

     ```
     deployedObject = AdminConfig.showAttribute(dep, "deployedObject")
     ```
4. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $deployedObject warClassLoaderPolicy
     ```

     Example output:

     ```
     {warClassLoaderPolicy MULTIPLE}
     ```
   - Using Jython:

     ```
     AdminConfig.show(deployedObject, 'warClassLoaderPolicy')
     ```

     Example output:

     ```
     '[warClassLoaderPolicy MULTIPLE]'
     ```
5. Modify the attributes of the configuration object with the **modify** command, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $deployedObject {{warClassLoaderPolicy SINGLE}}
     ```
   - Using Jython:

     ```
     AdminConfig.modify(deployedObject, [['warClassLoaderPolicy', 'SINGLE']])
     ```
6. Save the configuration changes.
7. Verify the changes that you made to the attribute value with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $deployedObject warClassLoaderPolicy
     ```

     Example output:

     ```
     {warClassLoaderPolicy SINGLE}
     ```
   - Using Jython:

     ```
     AdminConfig.show(deployedObject, 'warClassLoaderPolicy')
     ```

     Example output:

     ```
     '[warClassLoaderPolicy SINGLE]'
     ```

Modifying class loader modes for applications using scripting

You can modify class loader modes for an application with scripting and the wsadmin tool.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

About this task

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:
     ```
     set dep [$AdminConfig getid /Deployment:ivtApp/]
     ```
   - Using Jython:
     ```
     dep = AdminConfig.getid('/Deployment:ivtApp/')
     ```
3. Identify the deployed object and set it to the depObject variable. For example:
   - Using Jacl:
     ```
     set depObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:
     ```
     depObject = AdminConfig.showAttribute(dep, 'deployedObject')
     ```
4. Identify the class loader and set it to the classldr variable. For example:
   - Using Jacl:
     ```
     set classldr [$AdminConfig showAttribute $depObject classloader]
     ```
   - Using Jython:
     ```
     classldr = AdminConfig.showAttribute(depObject, 'classloader')
     ```
5. Show the current attribute values of the configuration object with the **showall** command, for example:
   - Using Jacl:
     ```
     $AdminConfig showall $classldr
     ```
     Example output:
     ```
     {libraries {}} {mode PARENT_FIRST}
     ```
   - Using Jython:
     ```
     print AdminConfig.showall(classldr)
     ```
     Example output:
     ```
     [libraries []] [mode PARENT_FIRST]
     ```
6. Modify the attributes of the configuration object with the **modify** command, for example:
   - Using Jacl:
     ```
     $AdminConfig modify $classldr {{mode PARENT_LAST}}
     ```

- Using Jython:

  ```
  AdminConfig.modify(classldr, [['mode', 'PARENT_LAST']])
  ```

7. Save the configuration changes.
8. Verify the changes that you made to the attribute value with the **showall** command, for example:
   - Using Jacl:

     ```
     $AdminConfig showall $classldr
     ```

     Example output:

     ```
     {libraries {}} {mode PARENT_LAST}
     ```

   - Using Jython:

     ```
     AdminConfig.showall(classldr)
     ```

     Example output:

     ```
     [libraries []] [mode PARENT_LAST]
     ```

Modifying the starting weight of applications using scripting

You can use the wsadmin tool and scripting to modify the starting weight of an application.

Before you begin

About this task

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

To modify the starting weight of an application, perform the following steps:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:

     ```
     set dep [$AdminConfig getid /Deployment:MyApp/]
     ```

   - Using Jython:

     ```
     dep = AdminConfig.getid("/Deployment:MyApp/")
     ```

3. Identify the deployed object and set it to the depObject variable. For example:
   - Using Jacl:

     ```
     set depObject [$AdminConfig showAttribute $dep deployedObject]
     ```

   - Using Jython:

     ```
     depObject = AdminConfig.showAttribute(dep, "deployedObject")
     ```

4. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $depObject startingWeight
     ```

     Example output:

     ```
     {startingWeight 1}
     ```

   - Using Jython:

     ```
     AdminConfig.show(depObject, 'startingWeight')
     ```

     Example output:

     ```
     [startingWeight 1]
     ```

5. Modify the attributes of the configuration object with the **modify** command, for example:

- Using Jacl:

  ```
  $AdminConfig modify $depObject {{startingWeight 2}}
  ```

- Using Jython:

  ```
  AdminConfig.modify(depObject, [['startingWeight', '2']])
  ```

6. Verify the changes that you made to the attribute value with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $depObject startingWeight
     ```

     Example output:

     ```
     {startingWeight 2}
     ```

   - Using Jython:

     ```
     AdminConfig.show(depObject, 'startingWeight')
     ```

     Example output:

     ```
     [startingWeight 2]
     ```

7. Save the configuration changes.

Configuring name space bindings using the wsadmin tool

Use this topic to configure name space bindings with the Jython or Jacl scripting languages and the wsadmin tool.

Before you begin

About this task

Use this task and the following examples to configure string, Enterprise JavaBeans (EJB), CORBA, or indirect name space bindings on a cell.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the cell and assign it to the cell variable.

   Using Jacl:

   ```
   set cell [$AdminConfig getid /Cell:mycell/]
   ```

   Example output:

   ```
   mycell(cells/mycell|cell.xml#Cell_1)
   ```

   Using Jython:

   ```
   cell = AdminConfig.getid('/Cell:mycell/')
   print cell
   ```

   You can change this example to configure on a node or server here.

3. Add a new name space binding on the cell. There are four binding types to choose from when configuring a new name space binding. They are string, EJB, CORBA, and indirect.
   - To configure a string type name space binding:

     Using Jacl:

     ```
     $AdminConfig create StringNameSpaceBinding $cell {{name binding1} {nameInNameSpace
     myBindings/myString} {stringToBind "This is the String value that gets bound"}}
     ```

     Example output:

     ```
     binding1(cells/mycell|namebindings.xml#StringNameSpaceBinding_1)
     ```

     Using Jython:

     ```
     print AdminConfig.create('StringNameSpaceBinding', cell, [['name', 'binding1'],
     ['nameInNameSpace', 'myBindings/myString'], ['stringToBind', "This is the String value that gets bound"]])
     ```

   - To configure an EJB type name space binding:

     Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2} {nameInNameSpace myBindings/myEJB}
{applicationNodeName mynode} {bindingLocation SINGLESERVER} {applicationServerName server1}
{ejbJndiName ejb/myEJB}}
```

Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name', 'binding2'],  ['nameInNameSpace',
'myBindings/myEJB'], ['applicationNodeName', 'mynode'],  ['bindingLocation', 'SINGLESERVER'],
['applicationServerName', 'server1'], ['ejbJndiName', 'ejb/myEJB']])
```

This example is for an EJB located in a server. For an EJB in a cluster, change the configuration example to:

Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2} {nameInNameSpace myBindings/myEJB}
{bindingLocation SERVERCLUSTER} {applicationServerName cluster1} {ejbJndiName ejb/myEJB}}
```

Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name','binding2'],
['nameInNameSpace','myBindings/myEJB'], ['bindingLocation','SERVERCLUSTER'],
['applicationServerName','cluster1'], ['ejbJndiName','ejb/myEJB']])
```

Example output:

```
binding2(cells/mycell|namebindings.xml#EjbNameSpaceBinding_1)
```

- To configure a CORBA type name space binding:

  Using Jacl:

```
$AdminConfig create CORBAObjectNameSpaceBinding $cell {{name binding3} {nameInNameSpace
myBindings/myCORBA}{corbanameUrl corbaname:iiop:somehost.somecompany.com:2809#stuff/MyCORBAOjbect}
{federatedContext false}}
```

  Example output:

```
binding3(cells/mycell|namebindings.xml#CORBAObjectNameSpaceBinding_1)
```

  Using Jython:

```
print AdminConfig.create('CORBAObjectNameSpaceBinding', cell, [['name', 'binding3'],['nameInNameSpace',
'myBindings/myCORBA'],['corbanameUrl','corbaname:iiop:somehost.somecompany.com:2809#stuff/MyCORBAOjbect'],
['federatedContext', 'false']])
```

- To configure an indirect type name space binding:

  Using Jacl:

```
$AdminConfig create IndirectLookupNameSpaceBinding $cell
{{name binding4} {nameInNameSpace myBindings/myIndirect} {providerURL
corbaloc::myCompany.com:9809/NameServiceServerRoot} {jndiName jndi/name/for/EJB}}
```

  Example output:

```
binding4(cells/mycell|namebindings.xml#IndirectLookupNameSpaceBinding_1)
```

  Using Jython:

```
print AdminConfig.create('IndirectLookupNameSpaceBinding', cell, [['name', 'binding4'],
['nameInNameSpace','myBindings/myIndirect'], ['providerURL', 'corbaloc::myCompany.com:9809/NameServiceServerRoot'],
['jndiName', 'jndi/name/for/EJB']])
```

4. Save the configuration changes.

WSScheduleCommands command group of the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications with the wsadmin tool. The commands and parameters in the WSScheduleCommands group can be used to create and manage scheduler settings in your configuration. Schedulers enable J2EE application tasks to run at a requested time.

The WSScheduleCommands command group for the AdminTask object includes the following commands:

deleteWSSchedule

The **deleteWSSchedule** command deletes the settings of a scheduler from the configuration.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

getWSSchedule

The **getWSSchedule** command returns the settings of the specified scheduler.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

listWSSchedules

The **listWSSchedules** command lists the scheduler.

Parameters and return values

-displayObjectNames
Set the value of this parameter to `true` to list the key set configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key set group name and management scope. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

modifyWSSchedule

The **modifyWSSchedule** command changes the settings of an existing scheduler.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

-frequency
The period of time in days to wait before checking for expired certificates. (Integer, optional)

-dayOfWeek
The day of the week to check for expired certificates. (Integer, optional)

-hour
The hour of the day to check for expired certificates. (Integer, optional)

-minute
The minute to check for expired certificates. Use this parameter with the hour parameter. (Integer, optional)

-nextStartDate
The next time, in seconds, to check for expired certificate. (Long, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

WSNotifierCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure deployed applications with the wsadmin tool. The commands and parameters in the WSNotifierCommands group can be used to create and manage notifications settings. WS-Notification enables Web services to use the ″publish and subscribe″ messaging pattern, creating a one-to-many message distribution pattern.

The WSNotifierCommands command group for the AdminTask object includes the following commands:

- "deleteWSNotifier"
- "getWSNotifier" on page 292
- "listWSNotifier" on page 292
- "modifyWSNotifier" on page 292

deleteWSNotifier

The **deleteWSNotifier** command deletes the settings of a notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

getWSNotifier

The **getWSNotifier** command displays the settings of a particular notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

listWSNotifier

The **listWSNotifier** command lists the notifier from the configuration.

Parameters and return values

-displayObjectNames
If you set the value of this parameter to true, this command returns all notification configuration objects within the scope. If you set the value of this parameter to false, this command returns a list of strings that contain the key set group name and the management scope. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

modifyWSNotifier

The **modifyWSNotifier** command changes the settings of an existing notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

-logToSystemOut
Set the value of this parameter to true if you want the certificate expiration information to log to system out. If not, set the value of this parameter to false. (Boolean, optional)

-sendEmail
Set the value of this parameter to true if you want to e-mail the certificate expiration information. If not, set the value of this parameter to false. (Boolean, optional)

-emailList
The list of e-mail addresses where you want to send certificate expiration information. Separate the values in the list with colons (:). (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

CoreGroupManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications. The commands and parameters in the CoreGroupManagement group can be used to create and manage core groups. A core group is a high availability domain that consists of a set of processes in the same cell that can directly establish high availability relationships. A cell must contain at least one core group.

The CoreGroupManagement command group for the AdminTask object includes the following commands:

createCoreGroup

The **createCoreGroup** command creates a new core group. The core group that you create contains no members.

Target object

None

Parameters and return values

-coreGroupName
The name of the core group that you are creating. (String required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroup {-coreGroupName MyCoreGroup} AdminConfig.save()
```

Optionally, you can use the following sample script to add a description for the new core group:

```
set core [$AdminConfig getid /Cell:myCell/CoreGroup:MyCoreGroup/] $AdminConfig
 modify $core {{description "My Description"}} $AdminConfig save
```

- Using Jython string:

```
AdminTask.createCoreGroup('[-coreGroupName MyCoreGroup]')
```

Optionally, you can use the following sample script to add a description for the new core group:

```
core = AdminConfig.getid('/Cell:myCell/CoreGroup:MyCoreGroup/')
 AdminConfig.modify(core, [['description', "This is my new description"]]) AdminConfig.save()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.createCoreGroup ('[-interactive]')
```

deleteCoreGroup

The **delete Core Group** command deletes an existing core group. The core group that you specify must not contain any members. You cannot delete the default core group.

Target object

None

Parameters and return values

-coreGroupName
The name of the existing core group that will be deleted. (String required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroup {-coreGroupName MyCoreGroup}
```

- Using Jython string:

```
AdminTask.deleteCoreGroup ('[-coreGroupName MyCoreGroup]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroup (['-coreGroupName', 'MyCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.deleteCoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroup ('[-interactive]')
```

doesCoreGroupExist

The **doesCore Group Exist** command returns a boolean value that indicates if the core group that you specify exists.

Target object

None

Parameters and return values

-coreGroupName
The name of the core group. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask doesCoreGroupExist {-coreGroupName MyCoreGroup}
```

- Using Jython string:

```
AdminTask.doesCoreGroupExist ('[-coreGroupName MyCoreGroup]')
```

- Using Jython list:

```
AdminTask.doesCoreGroupExist (['-coreGroupName', 'MyCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask doesCoreGroupExist {-interactive}
```

- Using Jython string:

```
AdminTask.doesCoreGroupExist ('[-interactive]')
```

- Using Jython list:

```
AdminTask.doesCoreGroupExist (['-interactive'])
```

getAllCoreGroupNames

The **getAll CoreGroup Names** command returns a string that contains the names of all of the existing core groups

Target object

None

Parameters and return values

- Parameters: None
- Returns: String array (String[ ])

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getAllCoreGroupNames
```

- Using Jython string:

```
AdminTask.getAllCoreGroupNames()
```

- Using Jython list:

```
AdminTask.getAllCoreGroupNames()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getAllCoreGroupNames {-interactive}
```

- Using Jython string:

```
AdminTask.getAllCoreGroupNames ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getAllCoreGroupNames (['-interactive'])
```

getCoreGroupNameForServer

The **getCore GroupName ForServer** command returns the name of the core group in which the server that you specify is currently a member.

Target object

None

Parameters and return values

-nodeName
The name of the node that contains the server. (String, required)

-serverName
The name of the server. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getCoreGroupNameForServer {-nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.getCoreGroupNameForServer ('[-nodeName myNode -server Name
 myServer]')
```

- Using Jython list:

```
AdminTask.getCoreGroupNameForServer (['-nodeName', 'myNode', '-serverName',
 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getCoreGroupName ForServer {-interactive}
```

- Using Jython string:

```
AdminTask.getCoreGroupName ForServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getCoreGroupName ForServer (['-interactive'])
```

getDefaultCoreGroupName

The **getDefault CoreGroup Name** command returns the name of the default core group.

Target object

None

Parameters and return values

- Parameters: None
- Returns: String

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getDefaultCoreGroupName
```

- Using Jython string:

```
AdminTask.getDefaultCoreGroupName()
```

- Using Jython list:

```
AdminTask.getDefaultCoreGroupName()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getDefaultCoreGroupName {-interactive}
```

- Using Jython string:

```
AdminTask.getDefaultCoreGroupName ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getDefaultCoreGroupName (['-interactive'])
```

moveClusterToCoreGroup

The **moveCluster ToCore Group** command moves all of the servers in a cluster that you specify from a core group to another core group. All of the servers in a cluster must be members of the same core group.

Target object

None

Parameters and return values

-source
The name of the core group that contains the cluster that you want to move. The core group must exist prior to running this command. The cluster that you specify must be a member of this core group. (String, required)

-target
The name of the core group where you want to move the cluster. (String, required)

-clusterName
The name of the cluster that you want to move. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask moveClusterToCoreGroup {-source OldCoreGroup -target NewCoreGroup
 -clusterName ClusterOne}
```

- Using Jython string:

```
AdminTask.moveClusterToCoreGroup ('[-source OldCoreGroup -target NewCoreGroup
 -clusterName ClusterOne]')
```

- Using Jython list:

```
AdminTask.moveClusterToCoreGroup (['-source', 'OldCoreGroup', '-target',
 'NewCoreGroup', '-clusterName', 'ClusterOne'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask moveClusterToCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.moveClusterToCoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.moveClusterToCoreGroup (['-interactive'])
```

moveServerToCoreGroup

The **moveServer ToCore Group** command moves a server to a core group that you specify. When the server is added to the core group that you specify, it is removed from the core group where it originally resided.

Target object

None

Parameters and return values

-source
The name of the core group that contains the server that you want to move. The core group must already exist with the server that you specify being a member of the core group. (String, required)

-target
The name of the core group where you want to move the server. The core group that you specify must exist prior to running the command. (String, required)

-nodeName
The name of the node that contains the server that you want to move. (String, required)

-serverName
The name of the server that you want to move. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask moveServerToCoreGroup {-source OldCoreGroup -target NewCoreGroup
 -nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.moveServerToCoreGroup ('[-source OldCoreGroup -target NewCoreGroup
 -nodeName myNode -server Name myServer]')
```

- Using Jython list:

```
AdminTask.moveServerToCore Group(['-source', 'OldCore Group', '-target',
 'NewCoreGroup', '-node Name', 'myNode', '-serverName', 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask moveServerTo CoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.moveServerTo CoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.moveServerTo CoreGroup (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

CoreGroupBridgeManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications using scripting. The commands and parameters in the CoreGroupBridgeManagement group can be used to create and manage core group access points, TCP inbound channel port, and bridge interfaces. A bridge interface specifies a particular node and server that runs the core group bridge service.

The CoreGroupBridgeManagement command group for the AdminTask object includes the following commands:

createCoreGroupAccessPoint

The createCoreGroupAccessPoint command creates a default core group access point for the core group that you specify and adds it to the default access point group. If the default access point group does not exist, the command creates a default access point group.

Target object

Core group bridge settings object for the cell. (ObjectName, required).

Required parameters

-coreGroupName
The name of the core group for which the core group access point will be created. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroupAccessPoint {-interactive}
```

- Using Jython:

```
AdminTask.createCoreGroupAccessPoint('-interactive')
```

deleteCoreGroupAccessPoints

The deleteCoreGroupAccessPoints command deletes all the core group access points that are associated with a group that you specify.

Target object

Core group bridge settings object for the cell. (ObjectName, required)

Required parameters

-coreGroupName
The name of the core group whose core group access points will be deleted. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroupAccessPoints (cells/rohitbuildCell01|coregroupbridge.xml#
 CoreGroupBridgeSettings_1) "-coreGroupName DefaultCoreGroup"
```

- Using Jython string:

```
AdminTask.deleteCoreGroupAccessPoints('cells/rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1',
 '[-coreGroupName DefaultCoreGroup]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroupAccessPoints('(cells/ rohitbuildCell01|coregroupbridge.xml#
 CoreGroupBridgeSettings_1)', ['-coreGroupName', 'DefaultCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroupAccessPoints {-interactive}
```

- Using Jython:

```
AdminTask.deleteCoreGroupAccessPoints('-interactive')
```

exportTunnelTemplate

The exportTunnelTemplate command exports a tunnel template and its associated children to a simple properties file.

Target object

None

Required parameters

-tunnelTemplateName
Specifies the name of the tunnel template to export. (String, required)

-outputFileName
Specifies the name of the properties file to create. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.exportTunnelTemplate('[-tunnelTemplateName tunnelTemplate1 -outputFileName tunnelTemplate1.props]')
```

- Using Jython list:

```
AdminTask.exportTunnelTemplate(['-tunnelTemplateName', 'tunnelTemplate1', '-outputFileName',
 'tunnelTemplate1.props'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.exportTunnelTemplate('-interactive')
```

getNamedTCPEndPoint

The getNamedTCPEndPoint command returns the port associated with the bridge interface that you specify. The port that is returned is the one that is specified on the TCP inbound channel of the transport channel chain for bridge interface that you specify.

Target object

The bridge interface object for which the port will be listed. (ObjectName, required)

Required parameters

None

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getNamedTCPEndPoint (cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2)
```

- Using Jython string:

```
AdminTask.getNamedTCPEndPoint('cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2')
```

- Using Jython list:

```
AdminTask.getNamedTCPEndPoint('cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getNamedTCPEndPoint {-interactive}
```

- Using Jython string:

```
AdminTask.getNamedTCPEndPoint('-interactive')
```

importTunnelTemplate

The importTunnelTemplate command imports a tunnel template and its children to the cell configuration.

Target object

None

Required parameters

-inputFileName
Specifies the name of the tunnel template file to import. (String, required)

-bridegeInterfaceNodeName
Specifies the name of the secure proxy node to use for the core group bridge interface. (String, required)

-bridegeInterfaceServerName
Specifies the name of the secure proxy server to use for the core bridge interface. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.importTunnelTemplate('[-inputFileName tunnelTemplate1.props
 -bridegeInterfaceNodeName secureProxyNode -bridegeInterfaceServerName mySecureProxyServer]')
```

- Using Jython list:

```
AdminTask.importTunnelTemplate(['-inputFileName', 'tunnelTemplate1.props',
 '-bridegeInterfaceNodeName', 'secureProxyNode', '-bridegeInterfaceServerName', 'mySecureProxyServer'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.importTunnelTemplate('-interactive')
```

listCoreGroups

The listCoreGroups command returns a collection of core groups that are related to the core group that you specify.

Target object

The name of the core group for which the related core groups will be listed. (String, required)

Required parameters

-cgBridgeSettings
The group bridge settings object for the cell. (ObjectName, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listCoreGroups DefaultCoreGroup "-cgBridgeSettings
 (cells/rohitbuildCell01|coregroupbridge.xml# CoreGroupBridgeSettings_1)"
```

- Using Jython string:

```
AdminTask.listCoreGroups('DefaultCoreGroup', '[-cgBridgeSetting (cells/
 rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1)]')
```

- Using Jython list:

```
AdminTask.listCoreGroups('DefaultCoreGroup', ['-cgBridgeSetting', '(cells/
 rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1)'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listCoreGroups {-interactive}
```

- Using Jython:

```
AdminTask.listCoreGroups('-interactive')
```

listEligibleBridgeInterfaces

The listEligibleBridgeInterfaces command returns a collection of node, server, and transport channel chain combinations that are eligible to become bridge interfaces for the specified core group access point.

Target object

The core group access point object for which bridge interfaces will be listed. (ObjectName, required)

Required parameters

None

Optional parameters

None

Example output

A set of bridge interfaces. (Set of String) Each bridge interface is represented by a combination of a node, a server and a DCS channel chain: <node *name*>, <server *name*>, <DCS Channel Chain *objectName*. For example, an element of the set returned by this command may look like the following: `rohitbuild dmgr DCS-Secure(cells/rohitbuildCell/nodes/rohitbuild/servers/dmgr|server.xml#Chain_4)`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listEligibleBridgeInterfaces
 CGAP_DCG_2(cells/rohitbuildCell01|coregroupbridge.xml#CoreGroupAccessPoint_1089636614062)
```

- Using Jython string:

```
AdminTask.listEligibleBridgeInterfaces('CGAP_DCG_2(cells/rohitbuildCell01|coregroupbridge.xml#
 CoreGroupAccessPoint_1089636614062)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listEligibleBridgeInterfaces {-interactive}
```

- Using Jython:

```
AdminTask.listEligibleBridgeInterfaces('-interactive')
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

CoreGroupPolicyManagement command group for the AdminTask object

You can use the Jython scripting language to configure and administer policies for high availability groups with the wsadmin tool. Use the commands and parameters in the CoreGroupPolicyManagement group to create, delete, and modify policies.

Use the following commands to define policies for high availability groups. Policies are defined at the core group level and apply only to matching high availability groups associated with the core group of interest.

- createAllActivePolicy
- createMOfNPolicy
- createNoOpPolicy
- createOneOfNPolicy
- createStaticPolicy

- deletePolicy
- modifyPolicy

createAllActivePolicy

The createAllActivePolicy command creates a high availability group policy that keeps each of the application components running on each server in the high availability group at all times.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group to associate with the new policy. (String, required)

-policyName
Specifies the name of the policy. (String, required)

Use the following guidelines to specify the policyName parameter:

- Specify valid characters, including numbers, letters, underscores, and spaces.
- Begin the policy name with a number or a letter.
- End the policy name with a number, letter, or underscore.

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, required)

Use the following guidelines to specify the matchCriteria parameter:

- Do not begin the match criteria with the underscore or period characters.
- Do not use the following characters: `\/,#$@:;"*?<>|=+&%'`
- You must specify a value. This parameter can not be null or empty.

Optional parameters

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createAllActivePolicy('-coreGroupName myCoreGroup -policyName myPolicy
 –matchCriteria "[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

- Using Jython list:

```
AdminTask.createAllActivePolicy('-coreGroupName', 'myCoreGroup', '-policyName',
  'myPolicy', '–matchCriteria', '"[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAllActivePolicy('-interactive')
```

createMOfNPolicy

The createMOfNPolicy command creates a high availability group policy that allows you to specify the number (M) of high availability group members to keep active if it is possible to do so. The number of active members must be greater than one and less than or equal to the number of servers in the high availability group.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group to associate with the new policy. (String, required)

-policyName
Specifies the name of the policy. (String, required)

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, required)

Optional parameters

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-failBack
Specifies whether work items assigned to the failing server are moved to the server that is designated as the most preferred server for the group if a failure occurs. This field only applies for M of N and One of N policies. (Boolean, optional)

-preferredOnly
Specifies whether group members are only activated on servers that are on the list of preferred servers for this group. This field only applies for M of N and One of N policies. (Boolean, optional)

-serversList
Specifies the members to prefer when activating a group member. The members must be part of the core group for which the policy applies. Specify the value of the serverList parameter in the format of `node/server`. (String[], optional)

-numActive
Specifies the number of the high availability group members to activate. This field only applies for the M of N policy. (Integer, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createMOfNPolicy('-coreGroupName myCoreGroup -policyName myPolicy
 –matchCriteria "[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

- Using Jython list:

```
AdminTask.createMOfNPolicy('-coreGroupName', 'myCoreGroup', '-policyName',
  'myPolicy', '–matchCriteria', '"[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createMOfNPolicy('-interactive')
```

createNoOpPolicy

The createNoOpPolicy command creates a high availability group policy that indicates that no high availability group members are made active.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group to associate with the new policy. (String, required)

-policyName
Specifies the name of the policy. (String, required)

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, required)

Optional parameters

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createNoOpPolicy('-coreGroupName myCoreGroup -policyName myPolicy
 –matchCriteria "[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

- Using Jython list:

```
AdminTask.createNoOpPolicy('-coreGroupName', 'myCoreGroup', '-policyName', 'myPolicy', '–matchCriteria',
   '"[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createNoOpPolicy('-interactive')
```

createOneOfNPolicy

The createOneOfNPolicy command creates a high availability group policy that keeps one member of the high availability group active at all times. This is used by groups that desire singleton failover. If a failure occurs, the high availability manager starts the singleton on another server.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group to associate with the new policy. (String, required)

-policyName
Specifies the name of the policy. (String, required)

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, required)

Optional parameters

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-failBack
Specifies whether work items assigned to the failing server are moved to the server that is designated as the most preferred server for the group if a failure occurs. This field only applies for M of N and One of N policies. (Boolean, optional)

-preferredOnly
Specifies whether group members are only activated on servers that are on the list of preferred servers for this group. This field only applies for M of N and One of N policies. (Boolean, optional)

-serversList
Specifies the members to prefer when activating a group member. The members must be part of the core group for which the policy applies. Specify the value of the serverList parameter in the format of `node/server`. (String[], optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createOneOfNPolicy('[-coreGroupName DefaultCoreGroup –policyName MySIBusPolicy –matchCriteria
 "[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]" -isAlive 120 –serversList
 WASnode01/server1;WASnode02/server2]')
```

- Using Jython list:

```
AdminTask.createOneOfNPolicy(['-coreGroupName', 'DefaultCoreGroup', '–policyName',
 'MySIBusPolicy', '–matchCriteria', '"[ [type WSAF_SIB][WSAF_SIB_BUS MyBus] ]"', '-isAlive', '120',
 '–serversList', 'WASnode01/server1;WASnode02/server2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createOneOfNPolicy('-interactive')
```

createStaticPolicy

The createStaticPolicy command creates a high availability group policy that allows you to statically define or configure the active members of the high availability group.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group to associate with the new policy. (String, required)

-policyName
Specifies the name of the policy. (String, required)

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, required)

-serversList
Specifies the members to prefer when activating a group member. The members must be part of the core group for which the policy applies. Specify the value of the serverList parameter in the format of `node/server`. (String[], optional)

Optional parameters

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createStaticPolicy('[-coreGroupName myCoreGroup -policyName myPolicy
 -matchCriteria "[[criteria1 value1][criteria2 value2]]" -serversList node/server1;node/server2;node/server3]')
```

- Using Jython list:

```
AdminTask.createStaticPolicy(['-coreGroupName', 'myCoreGroup', '-policyName',
 'myPolicy', '-matchCriteria', '"[[criteria1 value1][criteria2 value2]]"', '-serversList',
 'node/server1;node/server2;node/server3'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createStaticPolicy('-interactive')
```

deletePolicy

The deletePolicy command deletes a specific core group policy from the configuration.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group from which the system removes the policy. (String, required)

-policyName
Specifies the name of the policy to delete. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deletePolicy('-coreGroupName myCoreGroup -policyName myPolicy')
```

- Using Jython list:

```
AdminTask.deletePolicy('-coreGroupName', 'myCoreGroup', '-policyName', 'myPolicy')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deletePolicy('-interactive')
```

modifyPolicy

The modifyPolicy command modifies a specific core group policy of interest. You can use the modifyPolicy command to change one or many policy settings.

Target object

None.

Required parameters

-coreGroupName
Specifies the name of the core group that the policy of interest is associated with. (String, required)

-policyName
Specifies the name of the policy to modify. (String, required)

Optional parameters

-newPolicyName
Specifies a new name for the policy of interest. (String, optional)

-matchCriteria
Specifies one or more name and value pairs that the system uses to associate this policy with a high availability group. These pairs must match attributes that are contained in the name of a high availability group before this policy is associated with that group. (java.util.Properties, optional)

-isAlive
Specifies, in seconds, the interval of time at which the high availability manager checks the health of the active group members that are governed by this policy. If a group member has failed, the server on which the group member resides is restarted. (Integer, optional)

-quorum
Specifies whether quorum checking is enabled for a group governed by this policy. Quorum is a mechanism that can be used to protect resources that are shared across members of the group in the event of a failure. Quorum is an advanced hardware function and should not be enabled unless you thoroughly understand how to properly use this function. If not used properly, this function can cause data corruption. (Boolean, optional)

-description
Specifies a description for the core group policy. (String, optional)

-customProperties
Specifies additional custom properties for the core group policy. (java.util.Properties, optional)

-numActive
Specifies the number of the high availability group members to activate. This field only applies for the M of N policy. (Integer, optional)

-preferredOnly
Specifies whether group members are only activated on servers that are on the list of preferred servers for this group. This field only applies for M of N and One of N policies. (Boolean, optional)

-failBack
Specifies whether work items assigned to the failing server are moved to the server that is designated as the most preferred server for the group if a failure occurs. This field only applies for M of N and One of N policies. (Boolean, optional)

-serversList
Specifies the members to prefer when activating a group member. The members must be part of the core group for which the policy applies. Specify the value of the serverList parameter in the format of `node/server`. (String[], optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyPolicy('-coreGroupName myCoreGroup -policyName myPolicy
 -newPolicyName myPolicyRenamed')
```

- Using Jython list:

```
AdminTask.modifyPolicy('-coreGroupName', 'myCoreGroup', '-policyName', 'myPolicy',
 '-newPolicyName', 'myPolicyRenamed')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyPolicy('-interactive')
```

Chapter 6. Configuring servers with scripting

You can use the wsadmin tool to configure application servers in your environment. An application server configuration provides settings that control how an application server provides services for running applications and their components.

About this task

After installing the product, you might need to configure additional options for your application server. With the wsadmin tool, you can use the commands for the AdminTask and AdminConfig objects to retrieve configuration IDs and invoke operations on the objects to configure the application server. Alternatively, you can use the script library to perform specific operations to configure your application servers. The scripting library provides a set of procedures to automate the most common application server administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

You might need to complete one or more of the following tasks to configure your application server:

- Create servers. Use the commands in the ServerManagement command group for the AdminTask object or the AdminServerManagement script library to create a new application server, Web server, proxy server, or generic server.
- Configure the Java virtual machine to run in debug mode. Use the commands in the ServerManagement command group for the AdminTask object or the configureJavaVirtualMachine script in the AdminServerManagement script library to modify your Java virtual machine (JVM) configuration.
- Configure EJB containers. You can use the AdminConfig object or the configureEJBContainer script in the AdminServerManagement script library to configure Enterprise JavaBeans (EJB) containers in your configuration.
- Configure the Performance Monitoring Infrastructure. You can use the wsadmin tool to configure the Performance Monitoring Infrastructure (PMI) in your environment. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.
- Configure Object Request Broker (ORB) services. You can use the AdminConfig object or the configureORBService script in the AdminServerManagement script library to configure an ORB service in your environment. An ORB manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.
- Configure processes. You can use the AdminConfig object or the configureProcessDefintion script in the AdminServerManagement script library to configure processes in your application server configuration. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.
- Configure the runtime transaction service. Use the AdminControl object or the configureTransactionService script in the AdminServerManagement script library to configure transaction properties for servers. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.
- Set port numbers to the serverindex.xml file. You can use the AdminConfig object, AdminTask object, or the scripts in the AdminServerManagement script library to modify the port numbers specified in the `serverindex.xml` file. The end points of the `serverindex.xml` file are part of different objects in the configuration.

- Disable components. You can use the AdminConfig object or the configureStateManageable script in the AdminServerManagement script library to disable components by invoking operations. This topic describes how to disable the nameServer component of the product. You can modify the examples in this topic to disable other components.
- Disable the trace service.
- Configure servlet caching. You can configure servlet caching with scripting and the wsadmin tool. The dynamic cache service works within an application server JVM, intercepting calls to cacheable objects. For example, it intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.
- Modify variables.
- Increase the Java virtual machine heap size.

Creating a server using scripting

Use the commands in the ServerManagement command group for the AdminTask object or the AdminServerManagement script library to create a new application server, Web server, proxy server, or generic server.

Before you begin

There are three ways to complete this task. This topic uses the AdminConfig object and the commands for the AdminTask object to create a new server configuration. Alternatively, you can use the scripts in the AdminServerManagement script library to create an application server, Web server, proxy server, or generic server.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Obtain the configuration ID of the node object.

   The following example obtains the configuration ID of the node and assigns it to the *mynode* variable:
   - Using Jacl:

     ```
     set node [$AdminConfig getid /Node:mynode/]
     ```
   - Using Jython:

     ```
     node = AdminConfig.getid('/Node:mynode/')
     ```
3. Determine whether to use the AdminConfig or AdminTask object to create the server.
4. Create the server.
   - The following example uses the commands for the AdminTask object to create a server:

     Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask createApplicationServer mynode
       {-name test1 –templateName default}
       ```
     - Using Jython:

       ```
       AdminTask.createApplicationServer(mynode,
       ['-name', 'test1', '-templateName', 'default'])
       ```
   - The following example uses the AdminConfig object to create a server:

     Using the AdminConfig object:
     - Using Jacl:

       ```
       $AdminConfig create Server $node {{name myserv}
       {outputStreamRedirect {{fileName myfile.out}}}}
       ```
     - Using Jython:

       ```
       AdminConfig.create('Server', node, [['name', 'myserv'],
       ['outputStreamRedirect', [['fileName', 'myfile.out']]]])
       ```
5. Save the configuration changes.

6. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Configuring the Java virtual machine using scripting

Use the wsadmin tool to configure settings for a Java virtual machine (JVM). As part of configuring an application server, you might define settings that enhance the way your operating system uses of the Java virtual machine.

About this task

There are three ways to perform this task. Use the steps in this topic to use the setJVMDebugMode command for the AdminTask object or the AdminConfig object to modify your JVM configuration. Alternatively, you can use the configureJavaVirtualMachine Jython script in the AdminServerManagement script library to enable, disable, or configure the debug mode for the JVM. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM settings using the configureJavaVirtualMachine script:

```
AdminServerManagement.configureJavaVirtualMachine(nodeName, serverName, debugMode, debugArgs, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

The Java virtual machine (JVM) is an interpretive computing engine responsible for running the byte codes in a compiled Java program. The JVM translates the Java byte codes into the native instructions of the host machine. The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it. JVM settings are part of an application server configuration.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. There are two ways to complete this step. You can use the setJVMDebugMode command for the AdminTask object or the AdminConfig object to modify your JVM configuration. Choose one of the following configuration methods:
   - Using the AdminTask object:
     - Using Jacl:

```
$AdminTask setJVMDebugMode {-serverName server1 -nodeName node1 -debugMode true}
```

     - Using Jython:

```
AdminTask.setJVMDebugMode (['-serverName', 'server1', '-nodeName', 'node1', '-debugMode', 'true'])
```

   - Using the AdminConfig object:
     a. Identify the server and assign it to the server1 variable, as the following example demonstrates:
        - Using Jacl:

```
set server1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

        - Using Jython:

```
server1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print server1
```

          Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

b. Identify the JVM that belongs to the server of interest and assign it to the `jvm` variable, as the following example demonstrates:

– Using Jacl:

```
set jvm [$AdminConfig list JavaVirtualMachine $server1]
```

– Using Jython:

```
jvm = AdminConfig.list('JavaVirtualMachine', server1)
print jvm
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaVirtualMachine_1)
```

c. Modify the JVM to enable debugging, as the following example demonstrates:

– Using Jacl:

```
$AdminConfig modify $jvm {{debugMode true} {debugArgs "-Djava.compiler=NONE -Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"}}
```

– Using Jython:

```
AdminConfig.modify(jvm, [['debugMode', 'true'], ['debugArgs',  "-Djava.compiler=NONE -Xdebug
-Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"]])
```

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Configuring EJB containers using scripting

You can use the AdminConfig object or the wsadmin script library to configure Enterprise JavaBeans (EJB) containers in your configuration.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your EJB container configuration. Alternatively, you can use the configureEJBContainer Jython script in the AdminServerManagement script library to configure EJB containers. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure EJB containers using the configureEJBContainer script:

```
AdminServerManagement.configureEJBContainer(nodeName, serverName, ejbName, passivationDir, defaultDatasourceJNDIName)
```

For additional information and argument definitions, see the documentation for the AdminServerManagement script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server of interest.

   The following examples identify the application server and assign it to the `serv1` variable:

   - Using Jacl:

     ```
     set serv1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     serv1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print serv1
     ```

The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`serv1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the application server configuration
`getid`	AdminConfig command
`/Cell:`*`mycell`*`/Node:`*`mynode`*`/Server:`*`server1`*`/`	The hierarchical containment path of the configuration object
`Cell`	Object type
`mycell`	Optional name of the object
`Node`	Object type
`mynode`	Optional name of the object
`Server`	Object type
`server1`	Optional name of the object

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Identify the EJB container that belongs to the server.

   The following example identifies the EJB container for the server of interest and assigns it to the `ejbc1` variable:

   - Using Jacl:

     ```
     set ejbc1 [$AdminConfig list EJBContainer $serv1]
     ```

   - Using Jython:

     ```
     ejbc1 = AdminConfig.list('EJBContainer', serv1)
     print ejbc1
     ```

   The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`ejbc1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	The object that represents the application server configuration
`list`	AdminConfig command
`EJBContainer`	The object type The name of the object type that you specify is the one based on the XML configuration files and does not have to be the same name that the administrative console displays.
`serv1`	Evaluates to the ID of the server of interest

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
```

4. View each of the attributes of the EJB container.

   The following example displays the EJB container attributes but does not display nested attributes:

   - Using Jacl:

     ```
     $AdminConfig show $ejbc1
     ```

     Example output:

     ```
     {cacheSettings (cells/mycell/nodes/mynode/servers/
     server1|server.xml#EJBCache_1)}
     {components {}}
     {inactivePoolCleanupInterval 30000}
     {parentComponent (cells/mycell/nodes/mynode/servers/
     server1|server.xml#ApplicationServer_1)
     {passivationDirectory ${USER_INSTALL_ROOT}/temp}
     {properties {}}
     {services {(cells/mycell/nodes/mynode/servers/
     server1|server.xml#MessageListenerService_1)}
     {stateManagement (cells/mycell/nodes/mynode/servers/
     server1|server.xml#StateManageable_10)}
     ```

   - Using Jython:

     ```
     print AdminConfig.show(ejbc1)
     ```

     Example output:

     ```
     [cacheSettings (cells/mycell/nodes/myode/servers/
     server1|server.xml#EJBCache_1)]
     [components []]
     [inactivePoolCleanupInterval 30000]
     [parentComponent (cells/mycell/nodes/myode/servers/
     server1|server.xml#ApplicationServer_1)
     [passivationDirectory ${USER_INSTALL_ROOT}/temp]
     [properties []]
     [services [(cells/mycell/nodes/myode/servers/
     server1|server.xml#MessageListenerService_1)]
     [stateManagement (cells/mycell/nodes/mynode/servers/
     server1|server.xml#StateManageable_10)]
     ```

     The previous commands consist of the following elements:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`print`	Jython command
`AdminConfig`	The object that represents the application server configuration
`showall`	AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container

   The following example displays the EJB container attributes, including nested attributes:

   - Using Jacl:

     ```
     $AdminConfig showall $ejbc1
     ```

     Example output:

     ```
     {cacheSettings {{cacheSize 2053}
       {cleanupInterval 3000}}}
     {components {}}
     {inactivePoolCleanupInterval 30000}
     {parentComponent (cells/mycell/nodes/mynode/servers/
     server1|server.xml#ApplicationServer_1)}
     ```

```
{passivationDirectory ${USER_INSTALL_ROOT}/temp}
{properties {}}
{services {{{context (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)}
  {listenerPorts {}}
  {properties {}}
  {threadPool {{inactivityTimeout 3500}
    {isGrowable false}
    {maximumSize 50}
    {minimumSize 10}}}}}}
{stateManagement {{initialState START}
  {managedObject (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)}}}}
```

- Using Jython:

```
print AdminConfig.showall(ejbc1)
```

Example output:

```
[cacheSettings [[cacheSize 2053]
  [cleanupInterval 3000]]]
[components []]
[inactivePoolCleanupInterval 30000]
[parentComponent (cells/mycell/nodes/mynode/servers/
server1|server.xml#ApplicationServer_1)]
[passivationDirectory ${USER_INSTALL_ROOT}/temp]
[properties []]
[services [[[context (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)]
  [listenerPorts []]
  [properties []]
  [threadPool [[inactivityTimeout 3500]
    [isGrowable false]
    [maximumSize 50]
    [minimumSize 10]]]]]]
[stateManagement {{initialState START]
  [managedObject (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)]]]
```

The previous commands consist of the following elements:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`print`	Jython command
`AdminConfig`	The object that represents the application server configuration
`showall`	AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container

5. Modify the attributes.

   The following example modifies the enterprise bean cache settings and it includes nested attributes:

   - Using Jacl:

     ```
     $AdminConfig modify $ejbc1 {{cacheSettings
     {{cacheSize 2500} {cleanupInterval 3500}}}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(ejbc1, [['cacheSettings',
     [['cacheSize', 2500],  ['cleanupInterval', 3500]]]])
     ```

The previous commands consist of the following elements:

Element	Description
$	Jacl operator for substituting a variable name with its value
AdminConfig	The object that represents the application server configuration
modify	AdminConfig command
ejbc1	Evaluates to the ID of the enterprise bean container
cacheSettings	The attribute of modify objects
cacheSize	The attribute of modify objects
2500	The value of the cacheSize attribute
cleanupInterval	The attribute of modify objects
3500	The value of the cleanupInterval attribute

The following example modifies the cleanup interval attribute:

- Using Jacl:

```
$AdminConfig modify $ejbc1 {{inactivePoolCleanupInterval 15000}}
```

- Using Jython:

```
AdminConfig.modify(ejbc1, [['inactivePoolCleanupInterval', 15000]])
```

The previous commands consist of the following elements:

Element	Description
$	Jacl operator for substituting a variable name with its value
AdminConfig	The object that represents the application server configuration
modify	AdminConfig command
ejbc1	Evaluates to the ID of the enterprise bean container
inactivePoolCleanupInterval	The attribute of modify objects
15000	The value of the inactivePoolCleanupInterval attribute

6. Save the changes.

Configuring the Performance Monitoring Infrastructure using scripting

You can use the wsadmin tool to configure the Performance Monitoring Infrastructure (PMI) in your environment. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your server configuration. Alternatively, you can use the configurePerformanceMonitoringService Jython script in the AdminServerManagement script library to configure PMI. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure PMI settings using the configurePerformanceMonitoringService script:

```
AdminServerManagement.configurePerformanceMonitoringService(nodeName, serverName, enable, initialSpecLevel,
  otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server of interest.

   Use the AdminConfig object and the getid command to retrieve the configuration ID of the application server of interest, and assign it to the `s1` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('Cell:mycell/Node:mynode/Server:server1/')
     ```

   The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the product configuration
`getid`	AdminConfig command
`Cell`	Attribute
`mycell`	Value of the Cell attribute
`Node`	Attribute
`mynode`	Value of the Node attribute
`Server`	Attribute
`server1`	Value of the Server attribute

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

3. Identify the PMI service that belongs to the server.

   Use the AdminConfig object and the list command to identify the PMI service, and assign it to the `pmi` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set pmi [$AdminConfig list PMIService $s1]
     ```

   - Using Jython:

     ```
     pmi = AdminConfig.list('PMIService', s1)
     print pmi
     ```

   The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`pmi`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object that represents the application server configuration
`list`	AdminConfig command
`PMIService`	AdminConfig object
`s1`	Evaluates to the ID of the application server of interest

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
```

4. Modify the PMI configuration attributes.

   Use the AdminConfig object and the modify command to modify the PMI configuration attributes, as the following example demonstrates:

   - Using Jacl:

     ```
     $AdminConfig modify $pmi {{enable true} {statisticSet all}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(pmi, [['enable', 'true'], ['statisticSet','all']]
     ```

   This example enables PMI service and sets the specification levels for all of the components in the server. The following specification levels are valid for the components.

   Note: The specification levels are case-sensitive values.

Specification level	Description
none	No statistics are enabled.
basic	Statistics specified in Java Enterprise Edition (Java EE), as well as top statistics like CPU usage and live HTTP sessions are enabled. This set is enabled *out-of-the-box* and provides basic performance data about runtime and application components.
extended	Basic set plus key statistics from various application Sserver components like WLM and Dynamic caching are enabled. This set provides detailed performance data about various runtime and application components.
all	All statistics are enabled.
custom	Enable or disable statistics selectively.

5. Save the configuration changes.
6. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Limiting the growth of JVM log files using scripting

You can use scripting to configure the size of Java virtual machine (JVM) log files. JVM logs record events or information from a running JVM.

Before you begin

There are two ways to perform this task. This topic demonstrates how to use the AdminConfig object to modify your server configuration. Alternatively, you can use the configureJavaProcessLogs Jython script in the AdminServerManagement script library to configure the JVM log settings. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM log settings using the configureJavaProcessLogs script:

```
AdminServerManagement.configureJavaProcessLogs(nodeName, serverName, processLogRoot, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server of interest.

   Determine the configuration ID of the application server of interest and assign it to the server1 variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print s1
     ```

     where:

`set`	is a Jacl command
`s1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is the object type
`mycell`	is the name of the object that will be modified
`Node`	is the object type
`mynode`	is the name of the object that will be modified
`Server`	is the object type
`server1`	is the name of the object that will be modified
`print`	a Jython command

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

3. Identify the stream log of interest.

   Determine the stream log of interest and assign it to the log variable. The following example identifies the output stream log:

   - Using Jacl:

     ```
     set log [$AdminConfig showAttribute $s1 outputStreamRedirect]
     ```

   - Using Jython:

     ```
     log = AdminConfig.showAttribute(s1, 'outputStreamRedirect')
     ```

   The following example identifies the error stream log:

   - Using Jacl:

     ```
     set log [$AdminConfig showAttribute $s1 errorStreamRedirect]
     ```

   - Using Jython:

     ```
     log = AdminConfig.showAttribute(s1, 'errorStreamRedirect')
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#StreamRedirect_2)
   ```

4. List the current values of the stream log.

   Use the following example to display the current values of the stream log of interest:

- Using Jacl:

  ```
  $AdminConfig show $log
  ```

- Using Jython:

  ```
  AdminConfig.show(log)
  ```

Example output:

```
{baseHour 24}
{fileName ${SERVER_LOG_ROOT}/SystemOut.log}
{formatWrites true}
{maxNumberOfBackupFiles 1}
{messageFormatKind BASIC}
{rolloverPeriod 24}
{rolloverSize 1}
{rolloverType SIZE}
{suppressStackTrace false}
{suppressWrites false}
```

5. Modify the rotation policy for the stream log.

   The following example sets the rotation log file size to two megabytes:

   - Using Jacl:

     ```
     $AdminConfig modify $log {{rolloverSize 2}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(log, ['rolloverSize', 2])
     ```

   The following example sets the rotation policy to manage itself. It is based on the age of the file with the rollover algorithm loaded at midnight, and the log file rolling over every 12 hours:

   - Using Jacl:

     ```
     $AdminConfig modify $log {{rolloverType TIME}
     {rolloverPeriod 12} {baseHour 24}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(log, [['rolloverType', 'TIME']
     ['rolloverPeriod', 12] ['baseHour', 24]])
     ```

   The following example sets the log file to roll over based on both time and size:

   - Using Jacl:

     ```
     $AdminConfig modify $log {{rolloverType BOTH} {rolloverSize 2}
     {rolloverPeriod 12} {baseHour 24}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(log, [['rolloverType', 'BOTH'] ['rolloverSize', 2]
     ['rolloverPeriod', 12] ['baseHour', 24]])
     ```

6. Save the configuration changes.
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

ProxyManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage proxy configurations. Use the commands and parameters in the ProxyManagement group to configure proxy servers for Web modules.

The ProxyManagement command group for the AdminTask object includes the following commands:

- "createWebModuleProxyConfig"
- "deleteWebModuleProxyConfig" on page 326
- "getServerSecurityLevel" on page 326
- "setServerSecurityLevel" on page 327

createWebModuleProxyConfig

The createWebModuleProxyConfig command creates a proxy server configuration for a Web module.

Target object

Specify the deployment object that represents the application for which the system creates the Web module proxy configuration.

Required parameters

-deployedObjectProxyConfigName
Specifies the name of the Web module of interest. (String)

Optional parameters

-enableProxy
Specifies whether the system enables the proxy server. Specify `true` to enable the proxy server. (Boolean)

-transportProtocol
Specifies the protocol that the proxy server uses to communicate with the Web module. The valid values are `HTTP`, `HTTPS`, and `ClientProtocol`. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWebModuleProxyConfig myApplication {-deployedObjectProxyConfigName
 MyWebModule -enableProxy true -transportProtocol HTTPS}
```

- Using Jython string:

```
AdminTask.createWebModuleProxyConfig('myApplication', '[-deployedObjectProxyConfigName
 MyWebModule -enableProxy true -transportProtocol HTTPS]')
```

- Using Jython list:

```
AdminTask.createWebModuleProxyConfig(myApplication, ['-deployedObjectProxyConfigName',
 'MyWebModule', '-enableProxy', 'true', '-transportProtocol', 'HTTPS'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWebModuleProxyConfig {-interactive}
```

- Using Jython:

```
AdminTask.createWebModuleProxyConfig('-interactive')
```

deleteWebModuleProxyConfig

The deleteWebModuleProxyConfig command removes the proxy server configuration for a Web module.

Target object

Specify the deployment object that represents the application from which the system deletes the Web module proxy configuration.

Required parameters

-deployedObjectProxyConfigName
Specifies the name of the Web module of interest. (String)

Optional parameters

None

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteWebModuleProxyConfig myApplication {-deployedObjectProxyConfigName
 MyWebModule}
```

- Using Jython string:

```
AdminTask.deleteWebModuleProxyConfig('myApplication', '[-deployedObjectProxyConfigName
 MyWebModule]')
```

- Using Jython list:

```
AdminTask.deleteWebModuleProxyConfig(myApplication, ['-deployedObjectProxyConfigName',
 'MyWebModule'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteWebModuleProxyConfig {-interactive}
```

- Using Jython:

```
AdminTask.deleteWebModuleProxyConfig('-interactive')
```

getServerSecurityLevel

The getServerSecurityLevel command displays the current security level of the secure proxy server.

Target object

Specify the configuration ID of the secure proxy server of interest.

Optional parameters

-proxyDetailsFormat
Specifies the format of the details to display about the security level of the proxy server. Specify `levels` to display details as a security level for each setting. Specify `values` to display details as the actual setting for each proxy server. (String)

Sample output

The command returns the security level of the secure proxy server. If you specify the optional parameter, the command displays additional information about the security level of the server of interest.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getServerSecurityLevel myProxyServer {-proxyDetailsFormat levels}
```

- Using Jython string:

```
AdminTask.getServerSecurityLevel('myProxyServer', '[-proxyDetailsFormat
 levels]')
```

- Using Jython list:

```
AdminTask.getServerSecurityLevel(myProxyServer, ['-proxyDetailsFormat',
 'levels'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getServerSecurityLevel {-interactive}
```

- Using Jython:

```
AdminTask.getServerSecurityLevel('-interactive')
```

setServerSecurityLevel

The setServerSecurityLevel command modifies the server security level for a secure proxy server.

Target object

Specify the configuration ID of the secure proxy server of interest.

Optional parameters

-proxySecurityLevel
Specifies the level of security to apply to the proxy server. Valid values include `High`, `Medium`, and `Low`. (String)

You can also use this parameter to specify custom security settings by specifying the security setting ID and value, as defined in the following table:

ID	Description	Valid values
administration	Sets the administration security setting.	Specify `local` to allow local administration. Specify `remote` to allow remote administration.
routing	Sets the routing security setting. Using static routing specifies routing is performed through a flat configuration file using routing precedence that is inherent to the ordering of the directives. Requests can also be routed dynamically through a best match mechanism that determines the installed application or routing rule that corresponds to a specific request.	Specify `static` to use static routing, or specify `dynamic` to use dynamic routing.

ID	Description	Valid values
startupPermissions	Sets the startup permissions. The overall security level of the secured proxy server can be hardened by reverting the server process to run as an unprivileged user after startup. Although the secured proxy server must be started as a privileged user, changing the server process to run as an unprivileged user provides additional protection for local operating resources.	Specify `unprivileged` to run the server process as an unprivileged user, or specify `privileged` to run the server process as a privileged user.
errorPageHandling	Sets the error page handling. You can define a custom error page for each error code or a group of error codes on errors generated by the proxy server or the application server. This is done using HTTP status codes in responses to generate uniform customized error pages for the application. For security reasons, you can ensure that the error pages are read from the local file system instead of being forwarded to a custom remote application.	Specify `local` to read error pages from the local file system, or specify `remote` to allow the system to read error pages from remote applications.

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setServerSecurityLevel proxyServerID {-proxySecurityLevel
 administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local}
```

- Using Jython string:

```
AdminTask.setServerSecurityLevel('proxyServerID', '[-proxySecurityLevel
 administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local]'))
```

- Using Jython list:

```
AdminTask.setServerSecurityLevel(proxyServerID, ['-proxySecurityLevel',
 'administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setServerSecurityLevel {-interactive}
```

- Using Jython:

```
AdminTask.setServerSecurityLevel('-interactive')
```

Related concepts

DMZ Secure Proxy Server for IBM WebSphere Application Server administration options
The DMZ Secure Proxy Server for IBM WebSphere Application Server is administered differently than the WebSphere proxy server. The DMZ Secure Proxy Server for IBM WebSphere Application Server is a separate binary installed in the DMZ. Installing the DMZ Secure Proxy Server for IBM WebSphere Application Server in the DMZ requires that administration be managed differently for security reasons. Several administrative options are available for administering the DMZ Secure Proxy Server for IBM WebSphere Application Server to provide different levels of balance between security and usability.

DMZ Secure Proxy Server for IBM WebSphere Application Server routing considerations
This topic summarizes some of the security implications that must be considered when choosing how your DMZ Secure Proxy Server for IBM WebSphere Application Server will match incoming HTTP requests to an application or routing rule.

DMZ Secure Proxy Server for IBM WebSphere Application Server start up user permissions
The overall security level of the DMZ Secure Proxy Server for IBM WebSphere Application Server can be hardened by reverting the server process to run as an unprivileged user after startup. Although the DMZ Secure Proxy Server for IBM WebSphere Application Server must be started as a privileged user, changing the server process to run as an unprivileged user provides additional protection for local operating resources.

Error handling security considerations for the DMZ Secure Proxy Server for IBM WebSphere Application Server
The overall security level of the DMZ Secure Proxy Server for IBM WebSphere Application Server is partially determined by the choices made regarding the handling of custom errors.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"ServerManagement command group for the AdminTask object" on page 496
You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the ServerManagement group can be used to create and manage application server, Web server, proxy server, generic server and Java virtual machine (JVM) configurations.

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Configuring an ORB service using scripting

You can use the wsadmin tool to configure an Object Request Broker (ORB) service in your environment. An ORB manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your ORB configuration. Alternatively, you can use the configureORBService Jython script in the AdminServerManagement script library to configure settings for the ORB service. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM settings using the configureORBService script:

```
AdminServerManagement.configureORBService(nodeName, serverName, requestTimeout, requestRetriesCount,
requestRetriesDelay, connectionCacheMax, connectionCacheMin, locateRequestTimeout, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server and assign it to the server variable.

   Use the AdminConfig object and the getid command to retrieve the configuration ID of the server of interest, as the following example demonstrates:

   - Using Jacl:

```
set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print s1
```

   The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the WebSphere Application Server configuration
`getid`	AdminConfig command
`Cell`	Object type
`mycell`	Name of the object that will be modified
`Node`	Object type
`mynode`	Name of the object that will be modified
`Server`	Object type
`server1`	Name of the object that will be modified
`print`	Jython command

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Determine the ORB that belongs to the server.

   Use the AdminConfig object and the list command to identify the ORB that belongs to the server and assign it to the `orb` variable, as the following example demonstrates:

   - Using Jacl:

```
set orb [$AdminConfig list ObjectRequestBroker $s1]
```

   - Using Jython:

```
orb = AdminConfig.list('ObjectRequestBroker', s1)
print orb
```

   The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`orb`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the WebSphere Application Server configuration
`list`	AdminConfig command
`ObjectRequestBroker`	AdminConfig object

s1	Evaluates to the ID of server of interest
print	Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
```

4. Modify the ORB configuration attributes.

   The following example modifies the connection cache maximum and pass by value attributes. You can modify the example to change the value of other attributes.

   - Using Jacl:

```
$AdminConfig modify $orb {{connectionCacheMaximum 252} {noLocalCopies true}}
```

   - Using Jython:

```
AdminConfig.modify(orb, [['connectionCacheMaximum', 252], ['noLocalCopies',  'true']])
```

   The previous commands consist of the following elements:

Element	Description
$	Jacl operator for substituting a variable name with its value
AdminConfig	Object representing the WebSphere Application Server configuration
modify	AdminConfig command
orb	Evaluates to the ID of ORB
connectionCacheMaximum	Attribute
252	Value of the connectionCacheMaximum attribute
noLocalCopies	Attribute
true	Value of the noLocalCopies attribute

5. Save the configuration changes.
6. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

Configuring processes using scripting

You can use the wsadmin tool to configure processes in your application server configuration. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

About this task

There are three ways to perform this task. Complete the steps in this task to use the setProcessDefinition command for the AdminTask object or the AdminConfig object to modify your process definition configuration. Alternatively, you can use the configureProcessDefinition Jython script in the AdminServerManagement script library to configure process definition attributes. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure process definition settings using the configureProcessDefinition script:

```
AdminServerManagement.configureProcessDefintion(nodeName, serverName, otherParamList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Use the setProcessDefinition command for the AdminTask object or the AdminConfig object to modify your process definition configuration.
   - Use the following example to configure the process definition with the setProcessDefinition command for the AdminTask object:
     – Using Jacl:
     ```
     $AdminTask setProcessDefinition {-interactive}
     ```
     – Using Jython:
     ```
     AdminTask.setProcessDefinition (['-interactive'])
     ```
   - Use the following steps to configure the process definition with the AdminConfig option:
     a. Identify the server and assign it to the `s1` variable, as the following example demonstrates:
        – Using Jacl:
        ```
        set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
        ```
        – Using Jython:
        ```
        s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
        print s1
        ```
        The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object that represents the WebSphere Application Server configuration
`getid`	AdminConfig command
`Cell`	Object type
`mycell`	Name of the object that will be modified
`Node`	Object type
`mynode`	Name of the object that will be modified
`Server`	Object type
`server1`	Name of the object that will be modified
`print`	Jython command

        Example output:
        ```
        server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
        ```
     b. Identify the process definition for the server of interest, and assign it to the `processDef` variable, as the following example displays:
        – Using Jacl:
        ```
        set processDef [$AdminConfig list JavaProcessDef $s1]
        set processDef [$AdminConfig showAttribute $s1 processDefinitions]
        ```
        – Using Jython:
        ```
        processDef = AdminConfig.list('JavaProcessDef', s1)
        print processDef
        processDef = AdminConfig.showAttribute(s1, 'processDefinitions')
        ```
        Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#JavaProcessDef_1)
```

c. Modify the configuration attributes for the process definition.

The following example changes the working directory:

– Using Jacl:

```
$AdminConfig modify $processDef {{workingDirectory /home/myProfile/temp/user1}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['workingDirectory', '/home/myProfile/temp/user1']])
```

The following example modifies the name of the stderr file:

– Using Jacl:

```
set errFile [list stderrFilename \${LOG_ROOT}/server1/new_stderr.log]
set attr [list $errFile]
$AdminConfig modify $processDef [subst {{ioRedirect {$attr}}}]
```

– Using Jython:

```
errFile = ['stderrFilename', '\${LOG_ROOT}/server1/new_stderr.log']
attr = [errFile]
AdminConfig.modify(processDef, [['ioRedirect', [attr]]])
```

The following example modifies the process priority level:

– Using Jacl:

```
$AdminConfig modify $processDef {{execution {{processPriority 15}}}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['execution', [['processPriority', 15]]]])
```

The following example changes the maximum startup attempts. You can modify this example to change other attributes in the process definition object.

– Using Jacl:

```
$AdminConfig modify $processDef {{monitoringPolicy {{maximumStartupAttempts 1}}}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['monitoringPolicy', [['maximumStartupAttempts',  1]]]])
```

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Configuring the runtime transaction service using scripting

Use the wsadmin tool to configure transaction properties for servers. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.

About this task

There are two ways to perform this task. Use the steps in this task to use the AdminControl object to modify your transaction service configuration. Alternatively, you can use the configureTransactionService Jython script in the AdminServerManagement script library to configure the transaction service

configuration attributes. You can use the configureRuntimeTransactionService to update the transaction service MBean attributes. The wsadmin tool automatically loads the scripts when the tool starts.

Use the following syntax to configure transaction service settings using the configureTransactionService script:

```
AdminServerManagement.configureTransactionService(nodeName, serverName, totalTranLifetimeTimeout, clientInactivityTimeout,
  maximumTransactionTimeout, heuristicRetryLimit, heuristicRetryWait, propogateOrBMTTranLifetimeTimeout, asyncResponseTimeout,
  otherAttributeList)
```

Use the following syntax to configure runtime transaction service settings using the configureRuntimeTransactionService script:

```
AdminServerManagement.configureRuntimeTransactionService(nodeName, serverName, totalTranLifetimeTimeout,
  clientInactivityTimeout)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Identify the transaction service MBean for the application server.

   Use the completeObjectName command for the AdminControl object to return the transaction service MBean for the *server1* server, and to set it to the `ts` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set ts [$AdminControl completeObjectName cell=mycell,node=mynode,process=server1,type=TransactionService,*]
     ```

   - Using Jython:

     ```
     ts = AdminControl.completeObjectName('cell=mycell,node=mynode,process=server1,type=TransactionService,*')
     print ts
     ```

   The previous commands consist of the following elements:

Element	Description
`set`	A Jacl command
`ts`	A variable name
`$`	A Jacl operator for substituting a variable name with its value
`AdminControl`	An object that enables the manipulation of MBeans running in a server process
`completeObjectName`	An AdminControl command
`cell=mycell,node=mynode,process=server1,type=TransactionManagement`	A fragment of the object name whose complete name is returned by this command. It is used to find the matching object name which is, in this case, the transaction object MBean for the node *mynode*, where *mynode* is the name of the node that you use to synchronize configuration changes. For example: `type=TransactionService, process=server1`. It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, etc.

   Example output:

   ```
   WebSphere:cell=mycell,name=TransactionService,mbeanIdentifier=TransactionService,
   type=TransactionService,node=mynode,process=server1
   ```

2. Modify the runtime transaction service configuration attributes.

   - Using Jacl:

     ```
     $AdminControl setAttributes $ts {{clientInactivityTimeout 30} {totalTranLifetimeTimeout 180}}
     ```

   - Using Jython:

     ```
     AdminControl.setAttributes(ts, [['clientInactivityTimeout', 30],  ['totalTranLifetimeTimeout', 180]])
     ```

The previous commands consist of the following elements:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`AdminControl`	An object that enables the manipulation of MBeans running in a server process
`setAttributes`	An AdminControl command
`ts`	Evaluates to the ID of the transaction service of interest
`clientInactivityTimeout`	An attribute
`30`	The value of the clientInactivityTimeout attribute specified in seconds. A value of `0` means that there is no timeout limit.
`totalTranLifetimeTimeout`	An attribute
`180`	The value of the totalTranLifetimeTimeout attribute specified in seconds. A value of `0` means that there is no timeout limit.

Configuring the WS-Transaction specification level using the wsadmin tool

You can configure the default WS-Transaction specification level to use for outbound requests that include a Web Services Atomic Transaction (WS-AT) or Web Services Business Activity (WS-BA) coordination context.

About this task

The product supports both the WS-Transaction 1.1 and the WS-Transaction 1.0 specifications, but when an outbound request is sent, only one specification level can be used. The default WS-Transaction specification level is used if the specification level that the server requires cannot be determined from the provider policy (the WS-Transaction WS-Policy assertion). This situation might occur when the policy assertion is not available either from the WS-Transaction policy type of the client or from the WSDL of the target Web service. Also, this situation might occur when the policy assertion is available, but the client and the target Web service support both specification levels.

For details of the specifications, see Web Services Atomic Transaction support in the application server and Web Services Business Activity support in the application server.

You can set the default WS-Transaction specification level by using the wsadmin tool, as described in this task, or by using the administrative console and configuring the relevant transaction property for the application server.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the configuration ID of the transaction service. In Jacl, use the following code example:

```
set txService $AdminConfig list TransactionService
```

   In Jython, use the following code example:

```
txService = AdminConfig.list("TransactionService")
```

3. Modify the WSTransactionSpecificationLevel attribute to the value you require. In Jacl, to configure the server to use WS-Transaction 1.1, use the following code example:

```
$AdminConfig modify $txService {{WSTransactionSpecificationLevel WSTX_11}}
```

   In Jython, to configure the server to use WS-Transaction 1.0, use the following code example:

```
AdminConfig.modify ($txService,[["WSTransactionSpecificationLevel", "WSTX_10"]])
```

4. Save the configuration changes. See “Saving configuration changes with the wsadmin tool” on page 58.
5. Optional: In a network deployment environment only, synchronize the node. See “Synchronizing nodes with the wsadmin tool” on page 40.

Results

You have configured the default WS-Transaction specification level for the server.

Setting port numbers to the serverindex.xml file using scripting

You can use the wsadmin tool to modify the port numbers specified in the serverindex.xml file. The endpoints of the serverindex.xml file are part of different objects in the configuration.

About this task

There are multiple ways to complete this task. Complete the steps in this task to use the AdminConfig and AdminTask objects to configure ports in your environment. Alternatively, you can use the scripts in the AdminServerManagement script library to configure various ports in your configuration.

Before modifying ports using scripting, you must launch the wsadmin tool.

- Modify the BOOTSTRAP_ADDRESS attribute of the server1 process.

  The BOOTSTRAP_ADDRESS attribute is an attribute of the NameServer object that exists inside the server. The naming client uses the attribute to specify the naming server to look up the initial context. The following examples demonstrate how to modify the BOOTSTRAP_ADDRESS attribute:

  – Using the AdminTask object:

    - Using Jacl:

      ```
      $AdminTask modifyServerPort server1 {-nodeName mynode -endPointName BOOTSTRAP_ADDRESS
      -host myhost -port 2810}
      ```

    - Using Jython:

      ```
      AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName
      BOOTSTRAP_ADDRESS -host myhost -port 2810]')
      ```

  – Using the AdminConfig object. To modify its endpoint, obtain the ID of the NameServer object and issue a **modify** command, as the following example demonstrates:

    - Using Jacl:

      ```
      set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
      set ns [$AdminConfig list NameServer $s]
      $AdminConfig modify $ns {{BOOTSTRAP_ADDRESS {{port 2810} {host myhost}}}}
      ```

    - Using Jython:

      ```
      s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
      ns = AdminConfig.list('NameServer', s)
      AdminConfig.modify(ns, [['BOOTSTRAP_ADDRESS', [['host', 'myhost'],  ['port', 2810]]]])
      ```

- Modify the SOAP_CONNECTOR-ADDRESS attribute of the server1 process.

  The SOAP_CONNECTOR-ADDRESS attribute is an attribute of the SOAPConnector object that exists inside the server. HTTP transport uses the SOAP connector port for incoming SOAP requests. Use the following examples modify the SOAP_CONNECTOR-ADDRESS attribute:

  – Using the AdminTask object:

    - Using Jacl:

      ```
      $AdminTask modifyServerPort server1 {-nodeName mynode -endPointName SOAP_CONNECTOR_ADDRESS
      -host myhost -port 8881}
      ```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName SOAP_CONNECTOR_ADDRESS
-host myhost -port 8881]')
```

– To use the AdminConfig object to modify its endpoint, obtain the ID of the SOAPConnector object and issue a **modify** command, as the following example demonstrates:

  - Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set soap [$AdminConfig list SOAPConnector $s]
$AdminConfig modify $soap {{SOAP_CONNECTOR_ADDRESS {{host myhost} {port 8881}}}}
```

  - Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
soap = AdminConfig.list('SOAPConnector', s)
AdminConfig.modify(soap, [['SOAP_CONNECTOR_ADDRESS', [['host', 'myhost'],  ['port', 8881]]]])
```

- Modify the DRS_CLIENT_ADDRESS attribute of server1 process.

  The DRS_CLIENT_ADDRESS attribute is an attribute of the SystemMessageServer object that exists inside the server. The system uses this port to configure the Data Replication Service (DRS), which is a JMS-based message broker system for dynamic caching. The DRS_CLIENT_ADDRESS attribute is not available if a replication domain and a replicator entry have not been added to the server. Use the following examples configure the DRS_CLIENT_ADDRESS attribute:

  – Using the AdminTask object:

    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName
DRS_CLIENT_ADDRESS -host myhost -port 7874}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName
DRS_CLIENT_ADDRESS -host myhost -port 7874]')
```

  – To use the AdminConfig object to modify the endpoint of the DRS_CLIENT_ADDRESS attribute, obtain the ID of the SystemMessageServer object and issue a **modify** command, as the following example demonstrates:

    - Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set sms [$AdminConfig list SystemMessageServer $s]
$AdminConfig modify $sms {{DRS_CLIENT_ADDRESS {{host myhost} {port 7874}}}}
```

    - Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
sms = AdminConfig.list('SystemMessageServer', s)
AdminConfig.modify(sms, [['DRS_CLIENT_ADDRESS', [['host', 'myhost'], ['port', 7874]]]])
```

- Modify the JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes of the server1 process.

  The JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes are attributes of the JMSServer object that exists inside the server. The system uses these ports to configure the JMS provider topic connection factory settings. The following examples modify the JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes:

  – Using the AdminTask object:

    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName JMSSERVER_QUEUED_ADDRESS
-host myhost -port 5560}

$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName JMSSERVER_DIRECT_ADDRESS
-host myhost -port 5561}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName JMSSERVER_QUEUED_ADDRESS
-host myhost -port 5560]')

AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName JMSSERVER_DIRECT_ADDRESS
-host myhost -port 5561]')
```

– Using the AdminConfig object. To modify its endpoint, obtain the ID of the JMSServer object and issue a **modify** command, for example:

- Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set jmss [$AdminConfig list JMSServer $s]
$AdminConfig modify $jmss {{JMSSERVER_QUEUED_ADDRESS {{host myhost} {port 5560}}}}
$AdminConfig modify $jmss {{JMSSERVER_DIRECT_ADDRESS {{host myhost} {port 5561}}}}
```

- Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
jmss = AdminConfig.list('JMSServer', s)
AdminConfig.modify(jmss, [['JMSSERVER_QUEUED_ADDRESS', [['host', 'myhost'], ['port', 5560]]]])
AdminConfig.modify(jmss, [['JMSSERVER_DIRECT_ADDRESS', [['host', 'myhost'], ['port', 5561]]]])
```

- Modify the NODE_DISCOVERY_ADDRESS attribute of node agent process. The NODE_DISCOVERY_ADDRESS attribute is an attribute of the NodeAgent object that exists inside the server. The system uses this port to receive the incoming process discovery messages inside a node agent process. The following examples modify the NODE_DISCOVERY_ADDRESS attribute:
  – Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort nodeagent {-nodeName mynode -endPointName
NODE_DISCOVERY_ADDRESS -host myhost -port 7272}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('nodeagent', '[-nodeName mynode -endPointName
NODE_DISCOVERY_ADDRESS -host myhost -port 7272]')
```

  – Using the AdminConfig object. To modify its endpoint, obtain the ID of the NodeAgent object and issue a **modify** command, for example:
    - Using Jacl:

```
set nodeAgentServer [$AdminConfig getid /Cell:mycell/Node:mynode/Server:nodeagent/]
set nodeAgent [$AdminConfig list NodeAgent $nodeAgentServer]
$AdminConfig modify $nodeAgent {{NODE_DISCOVERY_ADDRESS {{host myhost} {port 7272}}}}
```

    - Using Jython:

```
nodeAgentServer = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:nodeagent/')
nodeAgent = AdminConfig.list('NodeAgent', nodeAgentServer)
AdminConfig.modify(nodeAgent, [['NODE_DISCOVERY_ADDRESS', [['host', 'myhost'], ['port', 7272]]]])
```

- Modify the CELL_DISCOVERY_ADDRESS attribute of deployment manager process. The CELL_DISCOVERY_ADDRESS attribute is an attribute of the deploymentManager object that exists inside the server. The system uses this port to receive the incoming process discovery messages inside a deployment manager process. Use the following examples modify the CELL_DISCOVERY_ADDRESS attribute:
  – Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort dmgr {-nodeName managernode -endPointName
CELL_MULTICAST_DISCOVERY_ADDRESS -host myhost -port 7272}

$AdminTask modifyServerPort dmgr {-nodeName managernode -endPointName
CELL_DISCOVERY_ADDRESS -host myhost -port 7278}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('dmgr', '[-nodeName managernode -endPointName
CELL_MULTICAST_DISCOVERY_ADDRESS -host myhost -port 7272]')

AdminTask.modifyServerPort ('dmgr', '[-nodeName managernode -endPointName
CELL_DISCOVERY_ADDRESS -host myhost -port 7278]')
```

  - To use the AdminConfig attribute to modify its endpoint, obtain the ID of the deploymentManager object and issue a **modify** command, for example:
    - Using Jacl:

```
set netmgr [$AdminConfig getid /Cell:mycell/Node:managernode/Server:dmgr/]
set deploymentManager [$AdminConfig list CellManager $netmgr]
$AdminConfig modify $deploymentManager {{CELL_MULTICAST_DISCOVERY_ADDRESS {{host myhost} {port 7272}}}}
$AdminConfig modify $deploymentManager {{CELL_DISCOVERY_ADDRESS {{host myhost} {port 7278}}}}
```

    - Using Jython:

```
netmgr = AdminConfig.getid('/Cell:mycell/Node:managernode/Server:dmgr/')
deploymentManager = AdminConfig.list('CellManager', netmgr)
AdminConfig.modify(deploymentManager, [['CELL_MULTICAST_DISCOVERY_ADDRESS', [['host', 'myhost'],
['port', 7272]]]])
AdminConfig.modify(deploymentManager, [['CELL_DISCOVERY_ADDRESS', [['host', 'myhost'], ['port', 7278]]]])
```

- Modify the WC_defaulthost attribute of the server1 process. Use the following examples modify WC_defaulthost endpoint:
  - Using the AdminConfig object:
    - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_defaulthost"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5555]]
           break
        }
     }
  }
}
```

    - Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_defaulthost":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5555]])
           break
```

- Modify the WC_defaulthost_secure attribute of the server1 process. Use the following examples to modify a WC_defaulthost_secure endpoint:
  - Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_defaulthost_secure
-host myhost -port 5544}
```

- Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_defaulthost_secure
-host myhost -port 5544]')
```

– Using the AdminConfig object:

- Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_defaulthost_secure"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5544]]
           break
        }
     }
  }
}
```

- Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_defaulthost_secure":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5544]])
           break
```

- Modify the WC_adminhost attribute of the server1 process.

  To modify a WC_adminhost endpoint, use one of the following examples:

  – Using the AdminTask object:

  - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_adminhost -host myhost -port 6666}
```

  - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_adminhost -host myhost -port 6666]')
```

  – Using the AdminConfig object:

  - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
```

```
            set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
            if {$endPointNm == "WC_adminhost"} {
               set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
               $AdminConfig modify $ePoint [list [list host myhost] [list port 6666]]
               break
            }
         }
      }
   }
```

- Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_adminhost":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 6666]])
           break
```

- Modify the WC_adminhost_secure attribute of server1 process.

  To modify a WC_adminhost_secure endpoint, use one of the following examples:

  – Using the AdminTask object:

    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_adminhost_secure -host myhost -port 5566}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_adminhost_secure
-host myhost -port 5566]')
```

  – Using the AdminConfig object:

    - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_adminhost_secure"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5566]]
           break
        }
     }
  }
}
```

    - Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
```

```
sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
sepList = sepString[1:len(sepString)-1].split(" ")
for specialEndPoint in sepList:
   endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
   if endPointNm == "WC_adminhost_secure":
      ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
      AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5566]])
      break
```

What to do next

Save the configuration changes.

Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

- Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

  ```
  AdminNodeManagement.syncActiveNodes()
  ```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

  ```
  AdminNodeManagement.syncNode("myNode")
  ```

Disabling components using scripting

You can disable components by invoking operations with scripting and the wsadmin tool. This topic describes how to disable the nameServer component of a configured server. You can modify the examples in this topic to disable other components.

About this task

There are two ways to complete this task. This topic uses the AdminConfig object to stop components in your environment. Alternatively, you can use the configureStateManageable script in the AdminServerManagement script library to enable and disable components. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure PMI settings using the configureStateManageable script:

```
AdminServerManagement.configureStateManageable(nodeName, serverName, parentType, initialState)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the server component and assign it to the nameServer variable.
   - Using Jacl:

```
set nameServer [$AdminConfig list NameServer $server]
```

   - Using Jython:

```
nameServer = AdminConfig.list('NameServer', server)
print nameServer
```

     Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
```

3. List the components belonging to the server.

   List the components that are associated with the server, and assign the components to the components variable, as the following example demonstrates:

   - Using Jacl:

```
set components [$AdminConfig list Component $server]
```

   - Using Jython:

```
components = AdminConfig.list('Component', server)
print components
```

The components variable contains a list of components.

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#ApplicationServer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#WebContainer_1)
```

4. Identify the nameServer component.

   Parse the components to identify the nameServer component, and assign it to the nameServer variable. Since the name server component is the third element in the list, retrieve this element by using an index of 2, as the following example demonstrates:

   - Using Jacl:

```
set nameServer [lindex $components 2]
```

   - Using Jython:

```
# get line separator
import  java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayComponents = components.split(lineSeparator)
nameServer = arrayComponents[2]
print nameServer
```

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
```

5. Disable the nameServer component.

   Modify the nested initialState attribute belonging to the stateManagement attribute to disable the nameServer component, as the following example demonstrates:

   - Using Jacl:

```
$AdminConfig modify $nameServer {{stateManagement {{initialState STOP}}}}
```

   - Using Jython:

```
AdminConfig.modify(nameServer, [['stateManagement', [['initialState', 'STOP']]]])
```

6. Save the configuration changes.
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Disabling the trace service using scripting

You can disable the services of a configured server with scripting and the wsadmin tool.

About this task

Perform the following steps to disable the trace service of a configured server. You can modify this example to disable a different service.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the server and assign it to the server variable. For example:

   - Using Jacl:

```
set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

- Using Jython:

```
server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print server
```

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. List all the services belonging to the server and assign them to the services variable. The following example returns a list of services:
   - Using Jacl:

```
set services [$AdminConfig list Service $server]
```

   - Using Jython:

```
services = AdminConfig.list('Service', server)
print services
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#AdminService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#DynamicCache_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#MessageListenerService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#RASLoggingService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#SessionManager_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#TransactionService_1)
```

4. Identify the trace service and assign it to the traceService variable.

   Since trace service is the seventh element in the list, retrieve this element by using index 6.
   - Using Jacl:

```
set traceService [$AdminConfig list TraceService $server]
```

   - Using Jython:

```
traceService = AdminConfig.list('TraceService', server)
print traceService
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
```

5. Disable the trace service by modifying the enable attribute. For example:
   - Using Jacl:

```
$AdminConfig modify $traceService {{enable false}}
```

   - Using Jython:

```
AdminConfig.modify(traceService, [['enable', 'false']])
```

6. Save the configuration changes.
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.
   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

     ```
     AdminNodeManagement.syncNode("myNode")
     ```

Configuring servlet caching with scripting

You can configure servlet caching with scripting and the wsadmin tool. The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. For example, it intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.

Before you begin

Before you can configure servlet caching, you must configure dynamic cache. Use the configureDynamicCache Jython script in the AdminServerManagement script library to configure dynamic caching. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure dynamic caching using the configureDynamicCache script:

```
AdminServerManagement.configureDynamicCache(nodeName, serverName, defaultPriority, cacheSize,
   externalCacheGroupName, externalCacheGroupType, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

About this task

After a servlet is invoked and completes generating the output to cache, a cache entry is created containing the output and the side effects of the servlet. These side effects can include calls to other servlets or JavaServer Pages (JSP) files or metadata about the entry, including timeout and entry priority information. Configure servlet caching to save the output of servlets and JavaServer Pages (JSP) files to the dynamic cache.

Note: If you use the wsadmin tool to enable servlet caching, verify that portlet fragment caching is also enabled. Similarly, if you use the wsadmin tool to disable servlet caching, verify that portlet fragment caching is also disabled. The settings for these two caching functions must remain synchronized. If you enable or disable servlet caching using the administrative console, synchronization performed automatically.

To see a list of parameters associated with dynamic caching, use the **attributes** command. For example:

```
$AdminConfig attributes DynamicCache
```

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the configuration ID of the server object.

   The following example sets the first server found to the `s1` variable:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('/Server:server1/')
     ```

3. Retrieve the Web containers for the server of interest. and assign them to the `wc` variable.

   The following example sets the Web container to the `wc` variable:

   - Using Jacl:

     ```
     set wc [$AdminConfig list WebContainer $s1]
     ```

   - Using Jython:

     ```
     wc = AdminConfig.list('WebContainer', s1)
     ```

4. Set a variable with the new value for the enableServletCaching attribute.

   Set the enableServletCaching attribute to `true` and assign it to the `serEnable` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set serEnable "{enableServletCaching true}"
     ```

   - Using Jython:

     ```
     serEnable = [['enableServletCaching', 'true']]
     ```

5. Enable dynamic caching.

   Use the AdminConfig object to modify the application server configuration, as the following example demonstrates:

- Using Jacl:

  ```
  $AdminConfig modify $wc $serEnable
  ```

- Using Jython:

  ```
  AdminConfig.modify(wc, serEnable)
  ```

Modifying variables using scripting

Use scripting and the wsadmin tool to modify variables in the application server.

About this task

Complete the following steps to modify an application server variable:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask setVariable {-interactive}
       ```

     - Using Jython:

       ```
       AdminTask.setVariable (['-interactive'])
       ```

   - Using the AdminConfig object. The following examples modify the DB2_JDBC_DRIVER_PATH variable on the node level:
     - Using Jacl:

       ```
       set varName DB2_JDBC_DRIVER_PATH
       set newVarValue C:/SQLLIB/java

       set node [$AdminConfig getid /Node:myNode/]

       set varSubstitutions [$AdminConfig list VariableSubstitutionEntry $node]

       foreach varSubst $varSubstitutions {
          set getVarName [$AdminConfig showAttribute $varSubst symbolicName]
          if {[string compare $getVarName $varName] == 0} {
             $AdminConfig modify $varSubst [list [list value $newVarValue]]
             break
          }
       }
       ```

     - Using Jython:

       ```
       varName = "DB2_JDBC_DRIVER_PATH"
       newVarValue = "C:/SQLLIB/java"

       node = AdminConfig.getid("/Node:myNode/")

       varSubstitutions = AdminConfig.list("VariableSubstitutionEntry",node).split
       (java.lang.System.getProperty("line.separator"))

       for varSubst in varSubstitutions:
          getVarName = AdminConfig.showAttribute(varSubst, "symbolicName")
          if getVarName == varName:
             AdminConfig.modify(varSubst,[["value", newVarValue]])
             break
       ```

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

```
AdminNodeManagement.syncNode("myNode")
```

Increasing the Java virtual machine heap size using scripting

Some servers might specify a Java virtual machine (JVM) heap size greater than the default. You can increase the heap size of the JVM using the administrative console, the wsadmin tool, or a Java client.

1. Set attributes that control the heap size for the JVM that is associated with the server.

   Use the administrative console or the wsadmin tool to control the heap size.
2. Increase the heap size of the JVM.

   Use the AdminTask object to modify the heap size, as the following examples demonstrate:

   - Using Jython:

     ```
     AdminTask.setJVMMaxHeapSize('-serverName server1 -nodeName node1 -maximumHeapSize heap_size')
     ```
   - Using Jacl:

     ```
     $AdminTask setJVMMaxHeapSize {-serverName server1 -nodeName node1 -maximumHeapSize heap_size}
     ```

PortManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure servers with the wsadmin tool. The commands and parameters in the PortManagement group can be used to list and modify application and server ports.

The PortManagement command group for the AdminTask object includes the following commands:

- “listApplicationPorts”
- “listServerPorts” on page 348
- “modifyServerPort” on page 348

listApplicationPorts

Use the **listApplicationPorts** command to list the ports in order to access a particular application.

Target object

The application name for which the list of ports is generated. (String)

Required parameters

None.

Return values

The ports that are used by the application that you specified.

Batch mode example usage

- Using Jacl:

  ```
  $AdminTask listApplicationPorts {}
  ```
- Using Jython string:

  ```
  AdminTask.listApplicationPorts ()
  ```

Interactive mode example usage

- Using Jacl:

  ```
  $AdminTask listApplicationPorts {-interactive}
  ```
- Using Jython string:

  ```
  AdminTask.listApplicationPorts ('[-interactive]')
  ```

listServerPorts

Use the **listServerPorts** command to list the ports that are used by the server that you specify.

Target object

The server name. (String)

Optional parameters

-nodeName
The name of the node. This parameter is only required when the server name is not unique in the cell. (String, optional)

Batch mode example usage

- Using Jacl:

  ```
  $AdminTask listServerPorts server1 {-nodeName myNode}
  ```
- Using Jython string:

  ```
  AdminTask.listServerPorts ('server1', '[-nodeName myNode]')
  ```

Interactive mode example usage

- Using Jacl:

  ```
  $AdminTask listServerPorts {-interactive}
  ```
- Using Jython string:

  ```
  AdminTask.listServerPorts ('[-interactive]')
  ```

modifyServerPort

Use the **modifyServer Port** command to modify the port that is used by the server.

Target object

The name of the server for which the port is modified.

Required parameters

-nodeName
The name of the server node. This parameter is required only if the server name is not unique in the cell. (String, required)

-endPointName
The name of the port to modify. (String, required)

Optional parameters

-host
The new value for the host name of the endpoint. (String, optional)

-port
The new value for the port number of the endpoint. (Integer, optional)

-modifyShared
Set this parameter to `true` to modify the port of interest if the port is shared between multiple transport channel chains. If this parameter is not specified, the command will not modify the port if it is used in more than one transport channel chain. (Boolean, optional)

Batch mode example usage

- Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName
port1 -port 5566 -modifyShared true}
```

- Using Jython string:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName port1
-port 5566 -modifyShared true]')
```

- Using Jython list:

```
AdminTask.modifyServerPort ('server1', ['-nodeName', 'myNode', '-endPointName',
'port1', '-port', '5566 -modifyShared true'])
```

Interactive mode example usage

- Using Jacl:

```
$AdminTask modifyServerPort {-interactive}
```

- Using Jython string:

```
AdminTask.modifyServerPort ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifyServerPort (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

DRS command group for the AdminTask object

You can use the Jython scripting language to configure the data replication service. Use the commands in the DRS command group display each configuration object that references a specific replication domain.

Use the following command to manage your data replication service:

- "listReplicationDomainReferences"

listReplicationDomainReferences

The **listReplicationDomainReferences** command displays a list of each configuration object that references a specific replication domain. Before deleting a replication domain, use this command to determine the configuration objects that are linked to the replication domain.

Target object

None.

Required parameters

-dataReplicationDomainName
Specifies the name of the data replication domain name of interest. (String, required)

Return value

The command displays each configuration object that references the data replication domain name of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.listReplicationDomainReferences('-dataReplicationDomainName myDataReplicationDomain')
```

- Using Jython list:

```
AdminTask.listReplicationDomainReferences('-dataReplicationDomainName', 'myDataReplicationDomain')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listReplicationDomainReferences('-interactive')
```

DynamicCache command group for the AdminTask object

You can use the Jython scripting language to manage the dynamic cache service. Use the commands in the DynamicCache command group to create object and service cache instances with the wsadmin tool.

Use the following commands to manage your dynamic cache configuration:

- "createObjectCacheInstance"
- "createServletCacheInstance" on page 351

createObjectCacheInstance

The **createObjectCacheInstance** command creates an object cache instance in your configuration. An object cache is a location where the dynamic cache stores, distributes, and shares data.

Target object

Specify the instance of the cache provider that the system stores the object cache instance objects under.

Required parameters

-name
Specifies the name of the object cache instance to create. (String, required)

-jndiName
Specifies the Java Naming and Directory Interface (JNDI) name for the object cache instance to create. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createObjectCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml#
#CacheProvider_1055745612404)','[-name objectName -jndiName myJNDI]')
```

- Using Jython list:

```
AdminTask.createObjectCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml#
#CacheProvider_1055745612404)',['-name', 'objectName', '-jndiName', 'myJNDI'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createObjectCacheInstance('-interactive')
```

createServletCacheInstance

The **createServletCacheInstance** command creates a servlet cache instance in your configuration. A servlet cache instance specifies the location where the dynamic cache stores, distributes, and shares data.

Target object

Specify the instance of the cache provider that the system stores the object cache instance objects under.

Required parameters

-name
Specifies the name of the servlet cache instance to create. (String, required)

-jndiName
Specifies the Java Naming and Directory Interface (JNDI) name for the servlet cache instance to create. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createServletCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml
l#CacheProvider_1055745612404)','[-name servletName -jndiName myJNDI]')
```

- Using Jython list:

```
AdminTask.createServletCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml
l#CacheProvider_1055745612404)',['-name', 'servletName', '-jndiName', 'myJNDI'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createServletCacheInstance('-interactive')
```

VariableConfiguration command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure servers with the wsadmin tool. The commands and parameters in the VariableConfiguration group can be used to remove variable definitions from the system, to set values for variables, or to query for variable values with a specific scope.

The VariableConfiguration command group for the AdminTask object includes the following commands:

- “removeVariable”
- “setVariable” on page 352
- “showVariables” on page 353

removeVariable

Use the **remove Variable** command to remove a variable definition from the system. A variable is a configuration property that you can use to provide a parameter for some values in the system.

Target object

None

Parameters and return values

-variableName
The name of the variable. (String, required)

-scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`

Supported types are Cell, Node, Servers, Application and Cluster, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask removeVariable {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.removeVariable ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.removeVariable (['-interactive'])
  ```

setVariable

Use the **set Variable** command to set the value for a variable. A variable is a configuration property that you can use to provide a parameter for some values in the system.

Target object

None

Parameters and return values

-variableName
The name of the variable. (String, required)

-scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

.

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

-variableValue
The value of the variable. (String, optional)

-variableDescription
The description of the variable. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:
  `$AdminTask setVariable {-variableName varname1 -scope Cell=localhost Node01Cell,Node= localhostNode01}`
- Using Jython string:
  `AdminTask.setVariable(' [-variableName varname1 -scope Cell=local hostNode01Cell,Node= localhostNode01]')`
- Using Jython list:
  `AdminTask.setVariable (['-variableName', 'varname1', '-scope', 'Cell=localhostNode 01Cell,Node=local hostNode01'])`

Interactive mode example usage:

- Using Jacl:
  `$AdminTask setVariable {-interactive}`
- Using Jython string:
  `AdminTask.setVariable ('[-interactive]')`
- Using Jython list:
  `AdminTask.setVariable (['-interactive'])`

showVariables

Use the **show Variables** command to list variable values under a scope.

Target object

None

Parameters and return values

- scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`

Supported types are Cell, Node, Servers, Application and Cluster, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

-variableName
The name of the variable. If you specify this parameter, the value of this variable is returned. If you do

not specify this parameter, all variables defined under the scope will return in list format where each element is a variable name and value pair. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showVariables {-interactive}
```

- Using Jython string:

```
AdminTask.showVariables ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showVariables (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 7. Setting up intermediary services using scripting

Use the wsadmin tool and the Jython scripting language to configure intermediary services, such as Web servers, proxy servers, and DataPower® appliances.

- Set up the DataPower appliance manager. Use the wsadmin tool to set up, query, and administer your DataPower appliance manager configurations. DataPower appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments.
- Regenerate the node plug-in configuration. You can use scripting and the wsadmin tool to regenerate the plug-in configuration for a Web server.
- Create virtual hosts using templates. Use scripting to create a new virtual host from a new or preexisting template. Virtual hosts let you manage a single application server on a single machine as if the application server were multiple application servers each on their own host machine.

Regenerating the node plug-in configuration using scripting

You can use scripting and the wsadmin tool to regenerate the node plug-in configuration.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to regenerate the node plug-in configuration:

1. Identify the plug-in and assign it to the `generator` variable, for example:

   Using Jython:

   ```
   generator = AdminControl.completeObjectName('type=PluginCfgGenerator,*')
   ```

   Using Jacl:

   ```
   set generator [$AdminControl completeObjectName type=PluginCfgGenerator,*]
   ```

   Additionally, you can specify the optional **node** parameter. In a Network Deployment environment, specify the node name of the deployment manager server.
2. Regenerate the node plug-in for a given Web server definition.

   Using Jython:

```
AdminControl.invoke(generator, 'generate', "profile_root/config mycell myWebServerNode myWebServerName true true")
```

   Using Jacl:

```
$AdminControl invoke $generator generate "profile_root/config mycell myWebServerNode myWebServerName true true"
```

Example

The following application-centric examples use the **generate**, **propagate**, and **propagateKeyring** operations for a given Web server definition:

Using Jython:

```
AdminControl.invoke(generator,'generate',"profile_root/config
01Cell03  01Node03 webserver1 true")

AdminControl.invoke(generator,'propagate',"profile_root/config
01Cell03  01Node03 webserver1")

AdminControl.invoke(generator,'propagateKeyring',"profile_root/config
01Cell03  01Node03 webserver1"
```

Using Jacl:

```
$AdminControl invoke $generator generate "profile_root/config  01Cell03  01Node03 webserver1 true"

$AdminControl invoke $generator propagate  "profile_root/config  01Cell03  01Node03 webserver1"

$AdminControl invoke $generator propagateKeyring "profile_root/config  01Cell03  01Node03 webserver1"
```

The following information explains the possible parameters that the **generate** operation accepts:

```
public void generate(java.lang.String
  configuration_root, java.lang.String myCellName,
 java.lang.String myNodeName, java.lang.String myServerName, java.lang.Boolean
 propagate, java.lang.Boolean propagateKeyring)
```

where:

configuration_root	is the root directory path of the particular configuration repository to be scanned.
myCellName	is the name of the cell in the configuration repository to restrict generation to.
myNodeName	is the name of the node in the configuration repository to restrict generation to.
myServerName	is the name of the server to restrict generation to.
propagate	is a boolean variable that specifies to propagate the configuration file.
propagateKeyring	is a boolean variable that specifies to propagate the keyring file.

The following network-centric example uses the **generate** operation to generate the plug-in configuration file for the cell:

Using Jython:

```
AdminControl.invoke(generator,'generate',"profile_root/config 01Cell03 null null plugin-cfg.xml")
```

Using Jacl:

```
$AdminControl invoke $generator generate "profile_root/config 01Cell03 null null plugin-cfg.xml"
```

The following information explains the possible parameters that the **generate** operation accepts:

```
public void generate(java.lang.String  app_server_root, java.lang.String
  configuration_root, java.lang.String myCellName,
 java.lang.String myNodeName, java.lang.String
 myServerName,java.lang.String myOutputFileName)
```

where:

app_server_root	is the root directory for the application server to run the command against.
configuration_root	is the root directory path for the configuration repository to be scanned.
myCellName	is the name of the cell in the configuration repository to restrict generation to.
myNodeName	is the name of the node in the configuration repository to restrict generation to.
myServerName	is the name of the server to restrict generation to.
myOutputFileName	is the path and filename of the generated plug-in configuration file.

Creating new virtual hosts using templates with scripting

Use scripting to create a new virtual host from a new or preexisting template.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Some configuration object types have templates that you can use when you create a virtual host. You can create a new virtual host using a preexisting template or by creating a new custom template. Perform the following steps to create a new virtual host using a template:

1. If you want to create a new custom template, perform the following steps:
   a. Copy and paste the following file into a new file, *myvirtualhostname.xml*:
      app_server_root/config/templates/default/virtualhosts.xml
   b. Edit and customize the new *myvirtualhostname.xml* file.
   c. Place the new file in the following directory:
      app_server_root/config/templates/custom/

   If you want the new custom template to appear with the list of templates, restart the deployment manager.

   The administrative console does not support the use of custom templates. The new template that you create will not be visible in the administrative console panels.
2. Use the AdminConfig object **listTemplates** command to list available templates, for example:
   - Using Jacl:

     ```
     $AdminConfig listTemplates VirtualHost
     ```
   - Using Jython:

     ```
     print AdminConfig.listTemplates('VirtualHost')
     ```

   Example output:

   ```
   default_host(templates/default:virtualhosts.xml#VirtualHost_1)
   my_host(templates/custom:virtualhostname.xml#VirtualHost_1)
   ```
3. Create a new virtual host. For example:
   - Using Jacl:

     ```
     set cell [$AdminConfig getid /Cell:NetworkDeploymentCell/]
     set vtempl [$AdminConfig listTemplates VirtualHost my_host]
     $AdminConfig createUsingTemplate VirtualHost $cell {{name newVirHost}} $vtempl
     ```
   - Using Jython:

     ```
     cell = AdminConfig.getid('/Cell:NetworkDeploymentCell/')
     vtempl = AdminConfig.listTemplates('VirtualHost', 'my_host')
     AdminConfig.createUsingTemplate('VirtualHost', cell, [['name', 'newVirHost']], vtempl)
     ```
4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Setting up the DataPower appliance manager using scripting

Use the application server and the wsadmin tool to set up, query, and administer your configured DataPower appliances in the DataPower appliance manager. DataPower appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments.

Before you begin

Before you begin, verify that each appliance that you want to manage has a 3.6.0.4 or higher level of firmware. Additionally, verify that the Appliance Management Protocol (AMP) endpoint is enabled for each appliance. If the XML Management interface AMP endpoint was disabled during installation, use the DataPower WebGUI to enable the AMP endpoint.

About this task

Use this topic to add DataPower appliances to the DataPower appliance manager, create managed sets, and assign appliances to managed sets in your environment. You can add as many or as few appliances and managed sets as you need.

The examples in this topic set up the DataPower appliance manager to administer two managed sets of DataPower appliances, update the appliance firmware, and configure domains. The first managed set represents a production environment that uses three DataPower appliances. The second managed set represents a test environment that uses one DataPower appliance. By setting up this configuration, you can use the second managed set in the test environment to modify and test the DataPower appliance settings before importing the test appliance domain to your production environment. Modify the examples to best configure your environment.

To view additional information and examples for the commands in this topic, refer to the documentation for the dpManagerCommands command group for the AdminTask object.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Add the DataPower appliances to the DataPower appliance manager configuration.

   Use the dpAddAppliance command to add appliances to the DataPower appliance manager. Each command invocation creates a task in the DataPower appliance manager and returns the task identifier as command output. The following commands add the `ProductionAppliance1`, `ProductionAppliance2`, `ProductionAppliance3`, and `TestAppliance1` appliances in your configuration and assign the task identifier to a specific variable:

```
app1Task=AdminTask.dpAddAppliance('[-hostname ProductionAppliance1.ibm.com -hlmPort
 5550 –name ProductionAppliance1 -userId admin -password mypassword]')
 app2Task=AdminTask.dpAddAppliance('[-hostname ProductionAppliance2.ibm.com -hlmPort 5550 –name
 ProductionAppliance2 -userId admin -password mypassword]') app3Task=AdminTask.dpAddAppliance('[-hostname
 ProductionAppliance3.ibm.com -hlmPort 5550 –name ProductionAppliance3 -userId admin -password
 mypassword]') testAppTask=AdminTask.dpAddAppliance('[-hostname TestAppliance1.ibm.com -hlmPort 5550 –name
 TestAppliance1 -userId admin -password mypassword]')
```

   The DataPower appliance manager submits tasks to add the appliances to the configuration. If the task uses a resource that another task is using, the system queues the new task until the other task is complete. Use the following example commands to monitor the status of the tasks:

```
param = '-taskId '+app1Task print AdminTask.dpGetTask(param)
```

   The command returns the task information and the appliance ID of the appliance as the value of the `result` attribute, as shown in the following sample output:

```
[ [currentStep 0] [totalSteps 0] [taskDescription [Add appliance ProductionAppliance1 to
 the DataPower appliance manager]] [currentStepTimestamp [Jan 18, 2008 2:32:25 PM]] [creationDate [Jan 18, 2008 2:32:23 PM]]
 [taskStatus 2] [taskId 1] [hasError false] [createdByUser defaultWIMFileBasedRealm/admin] [isComplete true] [result [00605
 20356]] ]
```

   After the system adds each appliance to your configuration, the appliances are unmanaged appliances. To manage each appliance, assign each appliance to a managed set.
3. Add the firmware version to the DataPower appliance manager.

Use the dpAddFirmwareVersion command to add the firmware version that the ProductionAppliance2 appliance uses to the DataPower appliance manager. The ProductionAppliance2 appliance is set as the master appliance later in this topic. Therefore, each appliance in the same managed set as the ProductionAppliance2 appliance will use the same firmware version.

```
firmwareTask=AdminTask.dpAddFirmwareVersion('[-file
 "C:\temp\dptestFW\dev-xs-143863-3_6_0_15.scrypt2" -userComment "my new firmware"]')
```

The command submits a task to the system to add the firmware, and assigns the task identifier to the firmwareTask variable.

4. Add managed sets in your DataPower appliance manager configuration.

   Use the dpAddManagedSet command to add managed sets to the DataPower appliance manager. Each command invocation creates a task in the DataPower appliance manager and returns the task identifier as command output. The following commands create the testSet and productionSet managed sets in your configuration:

```
AdminTask.dpAddManagedSet('-name testSet') AdminTask.dpAddManagedSet('-name
 productionSet')
```

   The command submits the task to the DataPower appliance manager and assigns the task identifier to the corresponding variables.

5. Verify that the system added each appliance and managed set to your configuration.

   The dpAddAppliance and dpAddManagedSet commands might not complete immediately. Before adding the appliances to the managed sets, verify that the system completed the tasks from the previous commands. If you did not set the command output to variables in the previous steps, use the following command to display each task identifier from the DataPower appliance manager:

```
AdminTask.dpGetAllTaskIds()
```

   Otherwise, use the dpGetTask command to determine whether the system has completed the tasks. Run the command for each task, as the following commands demonstrate:

```
AdminTask.dpGetTask('-taskId '+app1Task) AdminTask.dpGetTask('-taskId
 '+app2Task) AdminTask.dpGetTask('-taskId '+app3Task) AdminTask.dpGetTask('-taskId '+testAppTask)
 AdminTask.dpGetTask('-taskId '+prodSetTask) AdminTask.dpGetTask('-taskId '+firmwareTask)
 AdminTask.dpGetTask('-taskId '+prodSetTask)
```

   The commands return information about the asynchronous task of interest. The isComplete attribute displays a value of true if the task is complete. If it is not complete, note the value for the taskStatus attribute. If the returned value is 0, then the task is in a queue and the system has not started the task. If the returned value is 1, then the task is in progress. If the returned value is 2, then the task completed successfully. If the returned value is 3, then the task experienced an exception.

   For descriptions of the additional attributes that the command returns, see the documentation for the dpManagerCommands command group for the AdminTask object.

6. Assign the production appliances to the production managed set.

   To assign appliances to the managed set, you must know the appliance IDs of the appliances of interest. Use the dpGetAllApplianceIds command to display the appliance IDs of each appliance in your configuration, as the following example demonstrates:

```
AdminTask.dpGetAllApplianceIds()
```

   Use the dpGetAppliance command to display additional information for a specific appliance ID, as the following example demonstrates:

```
AdminTask.dpGetAppliance('[-applianceId "00605 20356"]')
```

   Use the dpManageAppliance command to add each appliance to the managed set, specifying the appliance ID of each appliance to add to the managed set.

   Note: To assign multiple appliances to a managed set, each appliance must be the same appliance type, such as XI50 or XS40, and model type. Additionally, the appliances must have the same required features installed. By ensuring that each appliance is exactly the same, you can copy the same firmware, domains, and settings for each appliance in the managed set.

   The following command examples add the ProductionAppliance1, ProductionAppliance2, and ProductionAppliance3 appliances to the productionSet managed set, and specify the

ProductionAppliance2 appliance as the master appliance. If you do not specify an appliance as the master appliance, the system automatically assigns the first appliance in the managed set as the master appliance.

```
manageTask1=AdminTask.dpManageAppliance('[-managedSetId productionSet -applianceId
 "00605 20351"]') manageTask2=AdminTask.dpManageAppliance('[-managedSetId productionSet -applianceId "00605
 20352" -asMaster]') manageTask3=AdminTask.dpManageAppliance('[-managedSetId productionSet -applianceId "00605
 20353"]')
```

The command submits a task to the system and sets the corresponding task identifiers to the manageTask1, manageTask2, and manageTask3 variables.

7. Assign the test appliance to the test managed set.

   The following command uses the dpManageAppliance command to add the TestAppliance1 appliance to the testSet managed set, specifying the appliance ID of the TestAppliance1 appliance:

```
manageTask4=AdminTask.dpManageAppliance('[-managedSetId testSet -applianceId
 "00605 20354"]')
```

   The command submits a task to the system and sets the corresponding task identifiers to the manageTask4 variable.

8. Verify that the system added the appliances to the managed sets.

   Use the dpGetTask command to determine the status of the manageTask1, manageTask2, manageTask3, and manageTask4 tasks before continuing to configure the DataPower appliance manager.

Results

A test environment managed set and a production environment managed set exist in the DataPower appliance manager configuration in this example. The test environment managed set manages the TestAppliance1 appliance. The production environment managed set manages the ProductionAppliance1, ProductionAppliance2, and ProductionAppliance3 appliances. Each appliance in both managed sets uses the default domain.

What to do next

You can use the DataPower WebGUI to set up domains for the testAppliance1 appliance in the test environment if the domains do not already exist. After configuring and testing the domains, you can use the wsadmin tool to copy the test environment appliance configuration to the production environment managed set.

You can also use the wsadmin tool to manage appliances, firmware, domains, managed sets, and appliance-specific settings. Additionally, the system creates versions of domains, firmware, and appliance-specific settings. You can use the wsadmin tool to modify the current version, or to revert to previous versions of domains, firmware, and appliance-specific settings.

Related concepts

WebSphere DataPower appliance manager overview
WebSphere DataPower appliance manager provides a set of capabilities for managing sets of appliances. DataPower appliance manager can be used to manage appliances with a 3.6.0.4 or higher level of firmware.

Secure Socket Layer communication with DataPower
Based on the default installations of the application server and the DataPower appliance manager, secure sockets layer (SSL) communication is used to send commands and receive events. The default SSL configuration used by the DataPower appliance manager can be strengthened by customizing the SSL connection. Modifying the default SSL configuration is optional and only needs to be done if the default configuration is not sufficient for your requirements.

Related tasks

"Copying DataPower appliance domains between managed sets using scripting"
Use the wsadmin tool to copy domains from one managed set to another, such as copying domains from a test environment to a production environment. Use the Datapower appliance manager and the wsadmin tool to manage appliances that are configured in the DataPower appliance manager.

"Updating firmware versions for DataPower appliances using scripting" on page 364
Use the wsadmin tool to update firmware for appliances within a managed set. Firmware version files from the manufacturer are specific to device types, model types, and libraries for features.

"Administering managed domains, firmware, and settings versions using scripting" on page 367
Use the wsadmin tool to administer managed domain and firmware version history for the DataPower appliance manager. You can revert to previous domain and firmware versions that exist in the DataPower appliance manager.

Using the DataPower appliance manager
The DataPower appliance manager automatically starts if you issue a request to the DataPower appliance manager and it is not already started. You can initiate a request using the wsadmin tool, or by selecting any of the administrative console pages that enable you to view or change settings for DataPower appliances, firmware, or managed sets, or the administrative console page that is used to monitor DataPower appliance manager tasks. The appliance manager also automatically starts when the deployment manager starts if there are any DataPower appliances configured in the appliance manager.

Adding DataPower appliances to the DataPower appliance manager
You can use the DataPower appliance manager that is provided with the product to administer a DataPower appliance. After you add an appliance to the DataPower appliance manager, you can make it part of a managed set of appliances if you want the DataPower appliance manager to keep the shared appliance settings for this appliance synchronized with the shared appliance settings of the other appliances that are part of that managed set.

Administering DataPower appliance domains
A DataPower appliance domain is a group of configuration information for an appliance. By default, these domains are unmanaged. You can use the administrative console to change an unmanaged domain to a managed domain, or to change a managed domain to an unmanaged domain. However, you cannot use the administrative console to configure a domain. You must use the DataPower WebGUI to configure a domain. See the DataPower WebGUI documentation for information about configuring a domain.

Related reference

"dpManagerCommands command group for the AdminTask object" on page 369
You can use the Jython scripting language to configure the DataPower appliance manager with the wsadmin tool. The IBM WebSphere DataPower appliance manager provides a set of capabilities for managing DataPower appliances. Use the commands and parameters in the dpManagerCommands group to query, configure, and administer the DataPower appliance manager.

Copying DataPower appliance domains between managed sets using scripting

Use the wsadmin tool to copy domains from one managed set to another, such as copying domains from a

test environment to a production environment. Use the Datapower appliance manager and the wsadmin tool to manage appliances that are configured in the DataPower appliance manager.

Before you begin

Before you begin, set up the DataPower appliance manager by adding and configuring appliances, managed sets, and firmware versions.

About this task

The examples in this topic refer to a DataPower appliance manager which administers two managed sets of DataPower appliances. The `productionSet` managed set represents a production environment that uses three DataPower appliances. The `testSet` managed set represents a test environment that uses one DataPower appliance. Use the DataPower WebGUI to configure and test the domains for the test environment managed set. Then, use this topic to copy the domains to your production environment. Modify the examples to best configure your environment.

To view additional information and examples for the commands in this topic, refer to the documentation for the dpManagerCommands command group for the AdminTask object.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the ID of the managed set.

   Use the dpGetAllManagedSetIDs command to display the IDs of each managed set in the DataPower appliance manager configuration, as the following command demonstrates:

   ```
   print AdminTask.dpGetAllManagedSetIds()
   ```

   Use the dpGetManagedSet command to display attributes for a given specific managed set ID, as the following example demonstrates:

   ```
   print AdminTask.dpGetManagedSet('-managedSetId testSet')
   ```
3. Determine the ID of the domain version to copy to the managed set.

   Use the dpGetAllMSDomainVersionIds command to display the IDs of each domain version for the BANKING domain, as the following example demonstrates:

   ```
   print AdminTask.dpGetAllMSDomainVersionIds('[-msDomainId testSet:BANKING]')"
   ```

   The command returns string array that contains the IDs of each domain within the managed set. You can optionally use the dpGetMSDomain command to display additional information for a specific domain ID.
4. Copy the domain version to the managed set.

   Use the dpCopyMSDomainVersion command to copy the domain version from the test environment to the production environment, as the following example demonstrates:

   ```
   copyTask=AdminTask.dpCopyMSDomainVersion('[-managedSetId productionSet -msDomainVersionId "testSet:BANKING:1"]')
   ```

   The command submits a task to the DataPower appliance manager and assigns the task identifier to the `copyTask` variable.
5. Verify that the system successfully copied the domain version to the managed set.

   Use the dpGetTask command to display the status and result information about the task, as the following example demonstrates:

   ```
   AdminTask.dpGetTask('-taskId copyTask')
   ```

   The commands return information about the asynchronous task of interest. The `isComplete` attribute displays a value of `true` if the task is complete. If it is not complete, note the value for the `taskStatus` attribute. If the returned value is `0`, then the task is in a queue and the system has not started the task. If the returned value is `1`, then the task is in progress. If the returned value is `2`, then the task completed successfully. If the returned value is `3`, then the task experienced an exception.

Results

The system uses the testSet:BANKING:1 version of the productionSet managed set in the production environment.

What to do next

You can use the DataPower WebGUI to configure additional domains.

You can use the commands in the dpManagerCommands command group and the wsadmin tool to manage appliances, firmware, domains, managed sets, and appliance-specific settings. Additionally, the system creates versions of domains, firmware, and appliance-specific settings. You can use the DataPower appliance manager and the wsadmin tool to modify the current version, or to revert to previous versions of domains, firmware, and appliance-specific settings.

Related concepts

WebSphere DataPower appliance manager overview
WebSphere DataPower appliance manager provides a set of capabilities for managing sets of appliances. DataPower appliance manager can be used to manage appliances with a 3.6.0.4 or higher level of firmware.

Related tasks

“Setting up the DataPower appliance manager using scripting” on page 358
Use the application server and the wsadmin tool to set up, query, and administer your configured DataPower appliances in the DataPower appliance manager. DataPower appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments.

“Updating firmware versions for DataPower appliances using scripting”
Use the wsadmin tool to update firmware for appliances within a managed set. Firmware version files from the manufacturer are specific to device types, model types, and libraries for features.

“Administering managed domains, firmware, and settings versions using scripting” on page 367
Use the wsadmin tool to administer managed domain and firmware version history for the DataPower appliance manager. You can revert to previous domain and firmware versions that exist in the DataPower appliance manager.

Administering DataPower appliance domains
A DataPower appliance domain is a group of configuration information for an appliance. By default, these domains are unmanaged. You can use the administrative console to change an unmanaged domain to a managed domain, or to change a managed domain to an unmanaged domain. However, you cannot use the administrative console to configure a domain. You must use the DataPower WebGUI to configure a domain. See the DataPower WebGUI documentation for information about configuring a domain.

Adding a new managed set
Add a new managed set to synchronize settings for multiple appliances. A managed set is a grouping of appliances that share the same shareable appliance settings, managed domains and firmware version. Shareable appliance settings and managed domains are propagated to the subordinate appliances from the master appliance.

Related reference

“dpManagerCommands command group for the AdminTask object” on page 369
You can use the Jython scripting language to configure the DataPower appliance manager with the wsadmin tool. The IBM WebSphere DataPower appliance manager provides a set of capabilities for managing DataPower appliances. Use the commands and parameters in the dpManagerCommands group to query, configure, and administer the DataPower appliance manager.

Updating firmware versions for DataPower appliances using scripting

Use the wsadmin tool to update firmware for appliances within a managed set. Firmware version files from the manufacturer are specific to device types, model types, and libraries for features.

Before you begin

Before you begin, set up the DataPower appliance manager by adding and configuring appliances and managed sets.

About this task

This topic provides an example for updating the firmware for multiple appliances that the DataPower appliance manager administers within a managed set. The appliances of interest are members of the `testSet` managed set. When updating the firmware on a managed set that manages multiple appliances, the system deploys the firmware version to the master appliance and then sequentially to each appliance in the managed set.

To view additional information and examples for the commands in this topic, refer to the documentation for the dpManagerCommands command group for the AdminTask object.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the IDs of the managed sets.

   Use the dpGetAllManagedSetIDs command to display the IDs of each managed set in the DataPower appliance manager configuration, as the following command demonstrates:

   ```
   AdminTask.dpGetAllManagedSetIds()
   ```

   You can optionally use the dpGetManagedSet command to display attributes for a given specific managed set ID, as the following example demonstrates:

   ```
   AdminTask.dpGetManagedSet('-managedSetId testSet')
   ```

3. Determine the firmware version to deploy to each managed set.

   You can use the DataPower appliance manager to deploy a new firmware version in your configuration or to revert to a previous firmware version that exist in your configuration.

   - To deploy a new firmware versions in your configuration, download the desired firmware versions from the DataPower Web site. You do not need to download new firmware versions if you are reverting to a previous firmware version that exists in your configuration. After downloading the new firmware versions, use the dpAddFirmwareVersion command to add the local firmware version to the DataPower appliance manager. The firmware version is associated with a specific firmware, which acts as a container for each firmware version that has the same appliance type, model type, and compatible features. The following command example adds a firmware version to the DataPower appliance manager:

     ```
     addFirmwareTask=AdminTask.dpAddFirmwareVersion('[-file "C:\temp\dptestFW\dev-xs-143863-3_6_0_16.scrypt2"
      -userComment "my new firmware for test"]')
     ```

     The command submits the tasks to the DataPower appliance manager and sets the task IDs to the corresponding variables. Use the following example commands to monitor the status of the tasks:

     ```
     param = '-taskId '+addFirmwareTask
     print AdminTask.dpGetTask(param)
     ```

     The command returns the task information and the firmware version ID as the value of the `result` attribute, as shown in the following sample output:

     ```
     [ [currentStep 0] [totalSteps 0] [taskDescription
      [Add new firmware version to the DataPower appliance manager]]
      [currentStepTimestamp [Jan 18, 2008 2:32:25 PM]] [creationDate [Jan 18, 2008 2:32:23 PM]]
      [taskStatus 2] [taskId 1] [hasError false] [createdByUser defaultWIMFileBasedRealm/admin]
      [isComplete true] [result [XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15]] ]
     ```

     You can use the dpGetAllFirmwareIds command to display the ID of each available firmware. Then, use the dpGetAllFirmwareVersionIDs command to get the IDs of each available firmware version of the firmware of interest. When you find the ID of the firmware version to use, set the value of the `result` attribute to a variable to use in the command that deploys the firmware version to the managed set, as the following example demonstrates:

     ```
     testFirmwareVersionID = 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15'
     ```

   - To deploy an existing firmware version to the managed set, use the following commands to determine the firmware versions of interest:
     - Use the dpGetAllFirmwareVersionIds command to display the version IDs for each firmware version of a specific firmware, as the following example demonstrates:

       ```
       AdminTask.dpGetAllFirmwareVersionIds('-firmwareId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;"')
       ```

     - For this example, the following two firmware versions exist:

       ```
       XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15
       XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.16
       ```

     - You can optionally use the dpGetFirmware command to display the appliance type, model type, strict features, and non-strict features for the firmware, as the following example demonstrates:

       ```
       AdminTask.dpGetFirmware('-firmwareId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;"')
       ```

– If you are unsure of which firmware to use, you can use the dpGetBestFirmware command to query for the firmware that matches your appliance type, model type, and appliance features. The command does not recommend a specific firmware. The command determines the firmware that matches your configuration. In the following example, the command queries for a firmware that contains the 3.6.0.4 firmware version:

```
AdminTask.dpGetBestFirmware('-applianceType "XS40" -modelType "9002" -applianceFeatures
 "JAXP-API" -level "3.6.0.4"')
```

4. Verify that the system added the new firmware version to the DataPower appliance manager.

   Use the getTask command to determine if the task is complete, as the following example demonstrates:

   ```
   AdminTask.dpGetTask('-taskId '+addFirmwareTask)
   ```

   The command returns information about the asynchronous task of interest. The `isComplete` attribute displays a value of `true` if the task is complete. If it is not complete, note the value for the `taskStatus` attribute. If the returned value is `0`, then the task is in a queue and the system has not started the task. If the returned value is `1`, then the task is in progress. If the returned value is `2`, then the task completed successfully. If the returned value is `3`, then the task experienced an exception.

5. Deploy the firmware version

   After the system adds the firmware version to the DataPower appliance manager, use the dpSetManagedSet command to assign the firmware version to the managed set. The following example assigns the newly downloaded firmware version to the test environment managed set:

   ```
   FVTask=AdminTask.dpSetManagedSet('-managedSetId testSet -desiredFirmwareVersionId '+testFirmwareVersionID)
   ```

   The command submits the tasks to the DataPower appliance manager and assigns the task identifiers to the corresponding variables. Deploying a firmware can take several minutes and will result in the appliance being restarted to run the new firmware version.

6. Verify that the system successfully assigned the new firmware versions to the managed sets.

   Use the getTask command to determine if the task is complete, as the following example demonstrates:

   ```
   AdminTask.dpGetTask('-taskId '+FVTask)
   ```

   The command returns information about the asynchronous task of interest. The `isComplete` attribute displays a value of `true` if the task is complete. If it is not complete, note the value for the `taskStatus` attribute. If the returned value is `0`, then the task is in a queue and the system has not started the task. If the returned value is `1`, then the task is in progress. If the returned value is `2`, then the task completed successfully. If the returned value is `3`, then the task experienced an exception.

7. Optional: Remove the firmware version that the system replaced from the DataPower appliance manager.

   If you do not want to keep a copy of the previous firmware version on the DataPower appliance manager, remove the firmware version from your configuration. You might want to keep the previous two firmware versions in case you need to revert to the previous version.

   Use the dpRemoveFirmwareVersion command to remove the firmware version that the managed set no longer uses. You can not remove firmware versions that are assigned to managed sets. The following example removes the firmware version from the DataPower appliance manager:

   ```
   AdminTask.dpRemoveFirmwareVersion('-firmwareVersionId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.13"')
   ```

Results

The managed set uses the new firmware version.

What to do next

You can use the wsadmin tool to manage appliances, firmware, domains, managed sets, and appliance-specific settings. Additionally, the system creates versions of domains, firmware, and appliance-specific settings. You can use the wsadmin tool to modify the current version, or to revert to

previous versions of domains, firmware, and appliance-specific settings.

Related concepts

WebSphere DataPower appliance manager overview
WebSphere DataPower appliance manager provides a set of capabilities for managing sets of appliances. DataPower appliance manager can be used to manage appliances with a 3.6.0.4 or higher level of firmware.

Related tasks

"Administering managed domains, firmware, and settings versions using scripting"
Use the wsadmin tool to administer managed domain and firmware version history for the DataPower appliance manager. You can revert to previous domain and firmware versions that exist in the DataPower appliance manager.

"Setting up the DataPower appliance manager using scripting" on page 358
Use the application server and the wsadmin tool to set up, query, and administer your configured DataPower appliances in the DataPower appliance manager. DataPower appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments.

"Copying DataPower appliance domains between managed sets using scripting" on page 361
Use the wsadmin tool to copy domains from one managed set to another, such as copying domains from a test environment to a production environment. Use the Datapower appliance manager and the wsadmin tool to manage appliances that are configured in the DataPower appliance manager.

Adding new firmware versions to the DataPower appliance manager
You can use the DataPower appliance manager to add a new firmware version to the DataPower appliance manager. Appliances that the DataPower appliance manager manages must have a 3.6.0.4 or higher firmware level.

Related reference

"dpManagerCommands command group for the AdminTask object" on page 369
You can use the Jython scripting language to configure the DataPower appliance manager with the wsadmin tool. The IBM WebSphere DataPower appliance manager provides a set of capabilities for managing DataPower appliances. Use the commands and parameters in the dpManagerCommands group to query, configure, and administer the DataPower appliance manager.

Administering managed domains, firmware, and settings versions using scripting

Use the wsadmin tool to administer managed domain and firmware version history for the DataPower appliance manager. You can revert to previous domain and firmware versions that exist in the DataPower appliance manager.

Before you begin

Before you begin, set up the DataPower appliance manager by adding and configuring appliances and managed sets.

About this task

When a DataPower administrator modifies the domain, firmware, or settings, the DataPower appliance manager automatically creates a copy of the previous configuration as domain, firmware, and settings versions if the following conditions apply:

- The DataPower appliance manager creates domain versions if the appliance is in a managed set and the domain is managed.
- The DataPower appliance manager only creates firmware versions when the administrator adds new firmware versions to the DataPower appliance manager.

- The DataPower appliance manager creates settings versions if the appliance is configured in a managed set.

You can use the wsadmin tool to view a history of versions, revert to a previous domains version, or copy a domain version to another managed set. This topic provides examples for modifying the domain, firmware, and settings versions. To view additional information and examples for the commands in this topic, refer to the documentation for the dpManagerCommands command group for the AdminTask object.

- Administer managed domain versions.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Display the IDs of the managed sets in your configuration.

     Use the dpGetAllManagedSetIDs command to display the IDs of each managed set in the DataPower appliance manager configuration, as the following command demonstrates:

```
AdminTask.dpGetAllManagedSetIds()
```

  3. Display each domain that is managed by a specific managed set.

     Use the dpGetAllMSDomainIds command to display the domain IDs for each domain in a specific managed set, as the following example demonstrates:

```
AdminTask.dpGetAllMSDomainIds('[-managedSetId myManagedSet]')
```

  4. Display each domain version that exists for a specific domain.

     Use the dpGetAllMSDomainVersionIds command to display the domain version IDs for each domain version that exists for a specific domain, as the following example demonstrates:

```
AdminTask.dpGetAllMSDomainVersionIds('[-msDomainId myManagedSet:domain1]')
```

     For this example, the command returns the following output:

```
[myManagedSet:domain1:1, myManagedSet:domain1:2, myManagedSet:domain1:3, myManagedSet:domain1:4]
```

  5. Display domain version information.

     Use the dpGetMSDomainVersion command to display detailed domain version information, including the time that the DataPower appliance manager created the version and comments, as the following example demonstrates:

```
AdminTask.dpGetMSDomainVersion('[-msDomainVersionId myManagedSet:domain1:1]')
```

  6. Specify a comment for a domain version.

     Use the dpSetDomainVersion command to specify a comment for a domain, as the following example demonstrates:

```
AdminTask.dpSetFirmwareVersion('-firmwareVersionId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15" -userComment "revert to firmware version 3.6.0.15"')
```

  7. Change to the first version of the domain.

     Use the dpSetMSDomainVersion command to modify the version of the domain that the managed set uses, as the following example demonstrates:

```
AdminTask.dpSetMSDomainVersion('[-msDomainVersionId myManagedSet:domain1:2 -userComment "revert to previous version"]')
```

- Administer firmware versions.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Display the IDs of the firmware in your configuration.

     Use the dpGetAllFirmwareIds command to display the IDs of each firmware in the DataPower appliance manager configuration, as the following command demonstrates:

```
AdminTask.dpGetAllFirmwareIds()
```

  3. Display the firmware versions that exist for a specific firmware.

     Use the dpGetAllFirmwareVersionIds command to display each firmware version ID for the firmware of interest, as the following example demonstrates:

```
AdminTask.dpGetAllFirmwareVersionIds('-firmwareId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;"')
```

     For this example, the command returns the following output:

```
[XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15, XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.16]
```

  4. Specify a comment for a firmware version.

Use the dpSetFirmwareVersion command to specify a comment for a firmware version, as the following example demonstrates:

```
AdminTask.dpSetFirmwareVersion('-firmwareVersionId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15" -userComment "revert to firmware version 3.6.0.15"')
```

5. Set the firmware version for the managed set.

   Use the dpSetManagedSet command to assign the firmware version to the managed set, as the following example demonstrates:

```
AdminTask.dpSetManagedSet('-managedSetId testSet -desiredFirmwareVersionId XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15')
```

- Administer settings versions.
  1. Display the IDs of the managed sets in your configuration.

     Use the dpGetAllManagedSetIDs command to display the IDs of each managed set in the DataPower appliance manager configuration, as the following command demonstrates:

```
AdminTask.dpGetAllManagedSetIds()
```

  2. Display additional information about the managed set of interest.

     Use the dpGetManagedSet command do display the configuration attributes for the managed set. The value of the `settingsId` attribute represents the settings object of the managed set of interest.

```
AdminTask.dpGetManagedSet('-managedSetId myManagedSet')
```

  3. Display each version of the settings of interest.

     Use the dpGetAllMSSettingsVersionIds command to display the ID of each settings version for the settings of interest, as the following example demonstrates:

```
AdminTask.dpGetAllMSSettingsVersionIds('-msSettingsId mySettings')
```

     For this example, the command returns the following output:

```
[myManagedSet:1, myManagedSet:2, MyManagedSet:3]
```

  4. Determine the settings version that the managed set currently uses.

     Use the dpGetMSSettings command to display the configuration attributes for the settings version that the managed set currently uses. In the command output, the value of the `desiredSettingsVersionId` attribute represents the settings version that the managed set uses.

```
AdminTask.dpGetMSSettings('-msSettingsId mySettings')
```

     For this example, the command returns the following output:

```
[MyManagedSet:3]
```

  5. Set the settings version that the managed set uses.

     Use the dpSetMSSettings command to set the settings version to use for the managed set, as the following example demonstrates:

```
AdminTask.dpSetMSSettings('-msSettingsId myMS1 -desiredSettingsVersionId myMS1:1')
```

  6. Specify a comment for a settings version.

     Use the dpSetMSSettingsVersion command to specify a comment for a settings version in your configuration, as the following example displays:

```
AdminTask.dpSetMSSettingsVersion('[-msSettingsVersionId myManagedSet:2 -userComment "added new timeserver"]')
```

What to do next

You can use the wsadmin tool to manage appliances, firmware, domains, managed sets, and appliance-specific settings.

dpManagerCommands command group for the AdminTask object

You can use the Jython scripting language to configure the DataPower appliance manager with the wsadmin tool. The IBM WebSphere DataPower appliance manager provides a set of capabilities for managing DataPower appliances. Use the commands and parameters in the dpManagerCommands group to query, configure, and administer the DataPower appliance manager.

IBM® WebSphere® DataPower SOA Appliances are purpose-built, easy-to-deploy network devices that simplify, help secure, and accelerate your XML and Web services deployments. The first time you use

DataPower appliance manager, no appliances, managed sets, or firmware versions exist. You must create appliances, add firmware versions, and create managed sets.

Before you begin, verify that each appliance that you want to manage has a 3.6.0.4 or higher level of firmware. Additionally, verify that the Appliance Management Protocol (AMP) endpoint is enabled for each appliance. If the XML Management interface AMP endpoint was disabled during installation, use the DataPower WebGUI to enable the AMP endpoint.

Note: The DataPower Web GUI is different from the DataPower appliance manager in the administrative console.

Use the following commands to administer the DataPower appliance manager:
- dpExport
- dpGetManager
- dpGetManagerStatus
- dpImport
- dpSetManager
- dpStopManager

Use the following commands to administer appliances:
- dpAddAppliance
- dpGetAllApplianceIds
- dpGetAllMSApplianceIds
- dpGetAppliance
- dpManageAppliance
- dpRemoveAppliance
- dpSetAppliance
- dpUnmanageAppliance

Use the following commands to administer managed sets:
- dpAddManagedSet
- dpGetAllManagedSetIds
- dpGetManagedSet
- dpRemoveManagedSet
- dpSetManagedSet
- dpSynchManagedSet

Use the following commands to administer firmware:
- dpAddFirmwareVersion
- dpGetAllFirmwareIds
- dpGetAllFirmwareVersionIds
- dpGetAllMSIdsUsingFirmwareVersion
- dpGetBestFirmware
- dpGetFirmware
- dpGetFirmwareVersion
- dpRemoveFirmwareVersion
- dpSetFirmwareVersion

Use the following commands to administer domains and domain versions:

- dpCopyMSDomainVersion
- dpGetAllDomainNames
- dpGetAllMSDomainIds
- dpGetAllMSDomainVersionIds
- dpGetMSDomain
- dpGetMSDomainVersion
- dpManageDomain
- dpRemoveMSDomainVersion
- dpSetMSDomain
- dpSetMSDomainVersion
- dpUnmanageDomain

Use the following commands to administer settings and settings versions:
- dpCopyMSSettingsVersion
- dpGetAllMSSettingsVersionIds
- dpGetMSSettings
- dpGetMSSettingsVersion
- dpRemoveMSSettingsVersion
- dpSetMSSettings
- dpSetMSSettingsVersion

Use the following commands to manage tasks:
- dpGetAllTaskIds
- dpGetTask
- dpPurgeTask

dpExport

The dpExport command exports the DataPower appliance manager configuration and versions.

Target object

None.

Required parameters

-file
Specifies the local system file to which the DataPower appliance manager exports the configuration and versions. (String, required)

Return value

The command returns the ID of the task that the system creates.

Batch mode example usage
- Using Jython string:

```
AdminTask.dpExport('-file
 /temp/DPManager.export')
```

- Using Jython list:

```
AdminTask.dpExport(['-file',
 '/temp/DPManager.export'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpExport('-interactive')
```

dpGetManager

The dpGetManager command displays the properties of the DataPower appliance manager.

Target object

None.

Return value

The command returns a properties object that contains the current settings of the DataPower appliance manager.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetManager()
```

dpGetManagerStatus

The dpGetManagerStatus command displays the status of the DataPower appliance manager.

Target object

None.

Return value

The command returns a description of the DataPower appliance manager status.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetManagerStatus()
```

dpImport

The dpImport command imports the DataPower appliance manager configuration and versions. The command replaces the existing configuration and versions with the imported configuration and versions.

Target object

None.

Required parameters

-file
Specifies the DataPower appliance manager system file that contains the configuration and versions to import. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpImport('-file
 /temp/DPManager.import')
```

- Using Jython list:

```
AdminTask.dpImport(['-file',
 '/temp/DPManager.export'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpImport('-interactive')
```

dpSetManager

The dpSetManager command modifies the DataPower appliance manager configuration.

Target object

None.

Optional parameters

-maxVersionsToStore
Specifies the new maximum number of versions to keep. (Integer, optional)

-versionsDirectory
Specifies the DataPower appliance manager system directory where the manager creates the versions. The command moves the existing versions from the current versions directory to the new versions directory. (String, optional)

Return value

If you specify a value for the versionsDirectory parameter, the command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetManager('-maxVersionsToStore 20 -versionsDirectory newDir')
```

- Using Jython list:

```
AdminTask.dpSetManager(['-maxVersionsToStore', '20', '-versionsDirectory', 'newDir'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetManager('-interactive')
```

dpStopManager

The dpStopManager command stops the DataPower appliance manager. The manager automatically restarts the next time the DataPower appliance manager is used.

Target object

None.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpStopManager()
```

dpAddAppliance

The dpAddAppliance command adds an appliance to the DataPower appliance manager.

Target object

None.

Required parameters

-hostname
Specifies the host name or IP address of the appliance. (String, required)

-hlmPort
Specifies the port number that the DataPower appliance manager uses to communicate to the appliance. The default value is `5550`. (Integer, required)

-name
Specifies the unique name of the appliance in the DataPower appliance manager. Do not specify the following characters within the name parameter: `\\/,#$@:;\"*?<>|=+&%'` (String, required)

-userId
Specifies the user ID that the DataPower appliance manager uses to access the appliance. (String, required)

-password
Specifies the base-64 encoded password that the DataPower appliance manager uses to access the appliance. (String, required)

Return value

The command returns the ID of the task that the system creates. When the task ends, the value of the result attribute in the task contains the ID of the new appliance.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpAddAppliance('[-hostname myappliance.ibm.com –name myappliance -userId
 admin -password mypassword]')
```

- Using Jython list:

```
AdminTask.dpAddAppliance(['-hostname', 'myappliance.ibm.com', '–name', 'myappliance', '-userId', '
  admin', '-password', 'mypassword'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpAddAppliance('-interactive')
```

dpGetAllApplianceIds

The dpGetAllApplianceIds command displays the ID of each DataPower appliance manager appliance.

Target object

None.

Return value

The command returns a string array that contains each appliance ID in your configuration. For appliances, the ID is the serial number of the DataPower appliance.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetAllApplianceIds()
```

dpGetAllMSApplianceIds

The dpGetAllMSApplianceIds command displays the IDs of each appliance in a DataPower appliance manager managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set of interest. (String, required)

Return value

The command returns a string array that contains the IDs of the appliances in the managed set of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllMSApplianceIds('[-managedSetId myManagedSet]')
```

- Using Jython list:

```
AdminTask.dpGetAllMSApplianceIds(['-managedSetId', 'myManagedSet'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllMSApplianceIds('-interactive')
```

dpGetAppliance

The dpGetAppliance command displays a specific DataPower appliance manager appliance.

Target object

None.

Required parameters

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to display. (String, required)

Return value

The command returns a properties object that contains the attributes of the appliance of interest, as the following sample output displays:

```
[ [userId admin] [applianceType XI50] [serialNumber [00605 20356]]
 [firmwareManagementStatus ManagementStatus.synced] [actualFirmwareLevel 3.6.1.0] [rollupOperationalStatus
 OperationalStatus.unknown] [hostname dp3.dyn.webahead.ibm.com] [settingsManagementStatus ManagementStatus.synced]
 [rollupManagementStatus ManagementStatus.synced] [applianceId [00605 20356]] [guiPort 8080] [modelType 9002] [isManaged true]
 [managedSetId jgMS1] [featureLicenses [MQ, TAM, DataGlue, JAXP-API, PKCS7-SMIME, WebSphere-JMS]] [hlmPort 5550] [isMaster true]
 [name dp3] ]
```

The following table provides descriptions of each attribute that the command returns:

Attribute	Description
applianceId	Displays the ID of the appliance of interest.
managedSetId	Displays the ID of the managed set for which the appliance is a member. If the appliance is not managed, the command does not return this attribute.
actualFirmwareLevel	Displays the level of the firmware of the appliance.
featureLicenses	Displays a list of feature entitlements for the appliance such as `MQ`, `TAM`, and so on.
guiPort	Displays the appliance port for the DataPower WebGUI interface.
hlmPort	Displays the port number for communication between the DataPower appliance manager and the appliance.
hostname	Displays the host name or internet protocol (IP) address of the appliance.
isManaged	Displays a value of `true` if the DataPower appliance manager manages the appliance.
isMaster	Displays a value of `true` if the appliance is the master appliance in the managed set. This property is not displayed if the appliance is not managed by the DataPower appliance manager.
modelType	Displays the model type of the appliance.
applianceType	Displays the appliance type.
settingsManagementStatus	Displays the management status of the settings on the appliance. This attribute is also referred to as the synchronization status on the DataPower WebGUI.
firmwareManagementStatus	Displays the management status of the firmware on the appliance. This attribute is also referred to as the synchronization status on the DataPower WebGUI.
domainManagementStatus	Displays the management status of a specific domain on the appliance. This attribute is also referred to as the synchronization status on the DataPower WebGUI.
domainOperationalStatus	Displays the operational status of a specific domain on the appliance.
rollupManagementStatus	Displays the aggregated management status of the firmware, settings, and domains on the appliance. This attribute is also referred to as the synchronization status on the DataPower WebGUI.
rollupOperationalStatus	Displays the aggregated operational status of the firmware, settings, and domains on the appliance.
serialNumber	Displays the serial number of the appliance of interest.
name	Displays the name of the appliance in the DataPower appliance manager.
userId	Displays the user ID that the DataPower appliance manager uses to access the appliance.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAppliance('-applianceId "00605 20356"')
```

- Using Jython list:

```
AdminTask.dpGetAppliance(['-applianceId', '00605 20356'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAppliance('-interactive')
```

dpManageAppliance

The dpManageAppliance command adds the appliance to a managed set and to start managing the appliance.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set for which the appliance is a member. (String, required)

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to add to the managed set. (String, required)

Optional parameters

-asMaster
Specifies whether to set the appliance as the master appliance of the managed set. The first appliance added to the managed set is set as the master appliance, even if this parameter is not specified. (String, optional)

Return value

The command returns the ID of the task that the command creates. When the task completes, the value of the result attribute is the ID of the appliance.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpManageAppliance('-managedSetId testMS1 -applianceId "00605 20351"')
```

- Using Jython list:

```
AdminTask.dpManageAppliance(['-managedSetId', 'testMS1', '-applianceId', '00605 20351'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpManageAppliance('-interactive')
```

dpRemoveAppliance

The dpRemoveAppliance command removes an appliance from the DataPower appliance manager. Also, the command removes the appliance from a managed set, if it is a member. You cannot remove an appliance that is a master in a managed set. You must select a different appliance as the master before removing a master appliance.

Target object

None.

Required parameters

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to remove from the managed set. For appliances, the ID is the serial number of the DataPower appliance. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpRemoveAppliance('[-applianceId "00605 20356"]')
```

- Using Jython list:

```
AdminTask.dpRemoveAppliance(['-applianceId', '00605 20356'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpRemoveAppliance('-interactive')
```

dpSetAppliance

The dpSetAppliance command modifies the DataPower appliance manager configuration for an appliance.

Target object

None.

Required parameters

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to modify. (String, required)

Optional parameters

-hostname
Specifies the host name or IP address of the appliance. (String, optional)

-hlmPort
Specifies the port number that the DataPower appliance manager uses to communicate to the appliance. The default value is `5550`. (Integer, optional)

-name
Specifies the unique name of the appliance in the DataPower appliance manager. Do not specify the following characters within the name parameter: `\\/,#$@:;\"*?<>|=+&%'` (String, optional)

-userId
Specifies the user ID that the DataPower appliance manager uses to access the appliance. (String, optional)

-password
Specifies the base-64 encoded password that the DataPower appliance manager uses to access the appliance. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetAppliance('[-applianceId "00605 20356" -hostname myappliance2.ibm.com
 -name myappliance2 -hlmPort 4500 -userId admin2 -password myPassword]')
```

- Using Jython list:

```
AdminTask.dpSetAppliance(['-applianceId', '00605 20356', '-hostname',
 'myappliance2.ibm.com', '-name', 'myappliance2', '-hlmPort', '4500', '-userId', 'admin2',
 '-password', 'myPassword'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetAppliance('-interactive')
```

dpUnmanageAppliance

The dpUnmanageAppliance command removes the appliance of interest from its managed set. The appliance is no longer managed, but remains defined to the manager.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set for which the appliance is a member. (String, required)

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to remove from the managed set. (String, required)

Return value

The command returns the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpUnmanageAppliance('[-managedSetId testMS1 -applianceId "00605
 20351"]')
```

- Using Jython list:

```
AdminTask.dpUnmanageAppliance(['-managedSetId', 'testMS1', '-applianceId "00605 20351"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpUnmanageAppliance('-interactive')
```

dpAddManagedSet

The dpAddManagedSet command adds a managed set to the DataPower appliance manager.

Target object

None.

Required parameters

-name
Specifies the unique name of the appliance in the DataPower appliance manager. Do not specify the following characters within the name parameter: \\/,#$@:;\"*?<>|=+&% (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpAddManagedSet('-name testMS')
```

- Using Jython list:

```
AdminTask.dpAddManagedSet(['-name testMS'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpAddManagedSet('-interactive')
```

dpGetAllManagedSetIds

The dpGetAllManagedSetIds command displays the IDs of each DataPower appliance manager managed set.

Target object

None.

Return value

The command returns a string array that contains each managed set ID.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetAllManagedSetIds()
```

dpGetManagedSet

The dpGetManagedSet command displays information for a specific DataPower appliance manager managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set of interest. (String, required)

Return value

The command returns a properties object that contains the attributes for the managed set. The following table provides additional information about the attributes that the command returns:

Attribute	Description
managedSetId	Displays the ID of the managed set.
masterApplianceId	Displays the ID of the master appliance of the managed set.
desiredFirmwareVersionId	Displays the ID of the firmware version to use for the managed set.
name	Displays the name associated with the managed set.
rollupOperationalStatus	Displays the operational status of the managed set.
rollupManagementStatus	Displays the management status of the managed set.
settingsId	Displays the ID of the settings for the managed set.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetManagedSet('-managedSetId testMS1')
```

- Using Jython list:

```
AdminTask.dpGetManagedSet(['-managedSetId', 'testMS1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetManagedSet('-interactive')
```

dpRemoveManagedSet

The dpRemoveManagedSet command removes a managed set from the DataPower appliance manager.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set to remove. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpRemoveManagedSet('-managedSetId  testMS')
```

- Using Jython list:

```
AdminTask.dpRemoveManagedSet(['-managedSetId', 'testMS'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpRemoveManagedSet('-interactive')
```

dpSetManagedSet

The dpSetManagedSet command modifies a DataPower appliance manager managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set to modify. (String, required)

Optional parameters

-masterApplianceId
Specifies the ID of the appliance to set as the master appliance for the managed set. This appliance must be a member of the managed set. This parameter is mutually exclusive with the desiredFirmwareVersionId parameter. (String, optional)

-desiredFirmwareVersionId
Specifies the ID of the firmware version to synchronize to each appliance in the managed set. This parameter is mutually exclusive with the masterApplianceId parameter. (String, optional)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetManagedSet('-managedSetId myManagedSet -masterApplianceId "00605
 20356"')
```

- Using Jython list:

```
AdminTask.dpSetManagedSet(['-managedSetId', 'myManagedSet', '-masterApplianceId', '"00605 20356"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetManagedSet('-interactive')
```

dpSynchManagedSet

The dpSynchManagedSet command manually synchronizes a DataPower appliance manager managed set. The manager automatically attempts to synchronize the member devices.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set to synchronize. (String, required)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSynchManagedSet('-managedSetId myManagedSet')
```

- Using Jython list:

```
AdminTask.dpSynchManagedSet(['-managedSetId', 'myManagedSet'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSynchManagedSet('-interactive')
```

dpAddFirmwareVersion

The dpAddFirmwareVersion command adds a firmware version to the DataPower appliance manager.

Target object

None.

Required parameters

-file
Specifies the file in the DataPower appliance manager file system that contains the firmware image to store in the firmware version. (String, required)

Optional parameters

-userComment
Specifies the comment to store in the firmware version. (String, optional)

Return value

The command returns the ID of the task that the command creates. When the task ends, the value of the result attribute displays the ID of the new firmware version.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpAddFirmwareVersion('[-file
 "/temp/dptestFW/dev-xs-143863-3_6_0_15.scrypt2" -userComment "my new firmware"]')
```

- Using Jython list:

```
AdminTask.dpAddFirmwareVersion(['-file',
 '"/temp/dptestFW/dev-xs-143863-3_6_0_15.scrypt2"', '-userComment', '"my new firmware"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpAddFirmwareVersion('-interactive')
```

dpGetAllFirmwareIds

The dpGetAllFirmwareIds command displays the IDs of each DataPower appliance manager firmware in the configuration.

Target object

None.

Return value

The command returns a string array for each firmware ID in the DataPower appliance manager configuration.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetAllFirmwareIds()
```

dpGetAllFirmwareVersionIds

The dpGetAllFirmwareVersionIds command displays the IDs of each DataPower appliance manager firmware version. A firmware version represents a firmware image you can deploy to a DataPower appliance.

Target object

None.

Required parameters

-firmwareId
Specifies the ID of the firmware to query. (String, required)

Return value

The command returns a string array that contains each firmware version in your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllFirmwareVersionIds('-firmwareId
 "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;"')
```

- Using Jython list:

```
AdminTask.dpGetAllFirmwareVersionIds(['-firmwareId',
 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllFirmwareVersionIds('-interactive')
```

dpGetAllMSIdsUsingFirmwareVersion

The dpGetAllMSIdsUsingFirmwareVersion command displays the IDs of the managed sets that use a firmware version.

Target object

None.

Required parameters

-firmwareVersionId
Specifies the ID of the firmware version of interest. (String, required)

Return value

The command returns a string array that contains the IDs of the managed sets that use the firmware version of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllMSIdsUsingFirmwareVersion('-firmwareVersionId
 "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15"')
```

- Using Jython list:

```
AdminTask.dpGetAllMSIdsUsingFirmwareVersion(['-firmwareVersionId',
 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllMSIdsUsingFirmwareVersion('-interactive')
```

dpGetBestFirmware

The dpGetBestFirmware command displays the firmware in the DataPower appliance manager that best matches the parameters. The firmware in the DataPower appliance manager contains one or more firmware versions that represent different versions of the firmware. This command does not identify the optimal firmware release from DataPower. The command identifies the firmware version that matches the appliance.

Target object

None.

Required parameters

-applianceType
Specifies the appliance type of the DataPower appliance for which to display the firmware. (String, required)

-modelType
Specifies the model type of the DataPower appliance for which to display the firmware. (String, required)

-applianceFeatures
Specifies the appliance features of the DataPower appliance for which to display the firmware. (String, required)

Optional parameters

-level
Specifies the level of the firmware version to contain in the returned firmware. (String, optional)

Return value

The command returns a properties object that contains the attributes of the firmware. The following table describes the attributes that the command returns:

Attribute	Description
firmwareId	Displays the firmware ID.
applianceType	Displays the appliance type for which the firmware is used.
modelType	Displays the model type of the firmware.
strictFeatures	Displays the strict features for which the firmware is used.
nonStrictFeatures	Displays the non strict features for which the firmware is used.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetBestFirmware('-applianceType "XS40" -modelType "9002"
 -applianceFeatures "JAXP-API" -level "3.6.0.4"')
```

- Using Jython list:

```
AdminTask.dpGetBestFirmware(['-applianceType', 'XS40', '-modelType', '9002',
 '-applianceFeatures', 'JAXP-API', '-level', '3.6.0.4'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetBestFirmware('-interactive')
```

dpGetFirmware

The dpGetFirmware command displays attributes for a specific DataPower appliance manager firmware.

Target object

None.

Required parameters

-firmwareId
Specifies the ID of the firmware of interest. (String, required)

Return value

The command returns a properties object that contains the attributes of the firmware of interest. The following table describes the attributes that the command returns:

Attribute	Description
firmwareId	Displays the firmware ID.
applianceType	Displays the appliance type for which the firmware is used.
modelType	Displays the model type of the firmware.
strictFeatures	Displays the strict features for which the firmware is used.
nonStrictFeatures	Displays the non strict features for which the firmware is used.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetFirmware('-firmwareId "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;"')
```

- Using Jython list:

```
AdminTask.dpGetFirmware(['-firmwareId', 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetFirmware('-interactive')
```

dpGetFirmwareVersion

The dpGetFirmwareVersion command displays a specific DataPower appliance manager firmware version. A firmware version represents a firmware image that you can deploy to a DataPower appliance.

Target object

None.

Required parameters

-firmwareVersionId
Specifies the ID of the firmware version of interest. (String, required)

Return value

The command returns a properties object that contains the attributes of the firmware version. The following table displays additional information about each attribute that the command returns:

Attribute	Description
firmwareVersionId	Displays the ID of the firmware version.
firmareId	Displays the ID of the firmware that contains the firmware version.
level	Displays the level of the firmware.
manufactureDate	Displays the date that the firmware was manufactured.
timestamp	Displays the date that the firmware image was loaded to the DataPower appliance manager.
userComment	Displays the comment stored with the firmware version.
isInUse	Displays whether a managed set uses the firmware version.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetFirmwareVersion('-firmwareVersionId
 "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15"')
```

- Using Jython list:

```
AdminTask.dpGetFirmwareVersion(['-firmwareVersionId',
 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetFirmwareVersion('-interactive')
```

dpRemoveFirmwareVersion

The dpRemoveFirmwareVersion command removes a firmware version from the DataPower appliance manager. The firmware that the system associates with the firmware version remains. Verify that a managed set is not currently using the firmware version to remove before you run this command.

Target object

None.

Required parameters

-firmwareVersionId
Specifies the ID of the firmware version to remove. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpRemoveFirmwareVersion('-firmwareVersionId
 "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15"')
```

- Using Jython list:

```
AdminTask.dpRemoveFirmwareVersion(['-firmwareVersionId',
 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpRemoveFirmwareVersion('-interactive')
```

dpSetFirmwareVersion

The dpSetFirmwareVersion command modifies a DataPower appliance manager firmware version.

Target object

None.

Required parameters

-firmwareVersionId
Specifies the ID of the firmware version to modify. (String, required)

Optional parameters

-userComment
Specifies the comment to store in the firmware version. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetFirmwareVersion('-firmwareVersionId
 "XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15" -userComment "new user comment"')
```

- Using Jython list:

```
AdminTask.dpSetFirmwareVersion(['-firmwareVersionId',
 'XS40:9002::DataGlue;JAXP-API;PKCS7-SMIME;HSM;:3.6.0.15', '-userComment', 'new user comment'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetFirmwareVersion('-interactive')
```

dpCopyMSDomainVersion

The dpCopyMSDomainVersion command copies a DataPower appliance manager managed domain version to a new managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set of interest. (String, required)

-msDomainVersionId
Specifies the ID of the managed domain version to copy. (String, required)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpCopyMSDomainVersion('[-managedSetId myManagedSet -msDomainVersionId "myManagedSet:default:1"]')
```

- Using Jython list:

```
AdminTask.dpCopyMSDomainVersion(['-managedSetId', 'myManagedSet', '-msDomainVersionId', 'myManagedSet:default:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpCopyMSDomainVersion('-interactive')
```

dpGetAllDomainNames

The dpGetAllDomainNames command display the names of each of the domains on a DataPower appliance.

Target object

None.

Required parameters

-applianceId
Specifies the ID of the appliance in the DataPower appliance manager to display. (String, required)

Optional parameters

-managed
Indicates whether to return the managed domains. The command returns the name of each domain on the appliance if you do not specify the managed or unmanaged parameters, or if you specify both parameters. (String, optional)

-unmanaged
Indicates whether the command returns the unmanaged domains. The command returns the name of each domain on the appliance if you do not specify the managed or unmanaged parameters, or if you specify both parameters. (String, optional)

Return value

The command returns a string array of the names of the domains in the appliance.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllDomainNames('[-applianceId "00605 20356" -managed -unmanaged]')
```

- Using Jython list:

```
AdminTask.dpGetAllDomainNames(['-applianceId', '00605 20356', '-managed',
 '-unmanaged'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllDomainNames('-interactive')
```

dpGetAllMSDomainIds

The dpGetAllMSDomainIds command displays the IDs of each domain in a DataPower appliance manager managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set of interest. (String, required)

Return value

The command returns a string array that contains the IDs of the domains in the managed set.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllMSDomainIds('[-managedSetId myManagedSet]')
```

- Using Jython list:

```
AdminTask.dpGetAllMSDomainIds(['-managedSetId', 'myManagedSet'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllMSDomainIds('-interactive')
```

dpGetAllMSDomainVersionIds

The dpGetAllMSDomainVersionIds command displays the ID of each domain version for a domain in a DataPower appliance manager managed set.

Target object

None.

Required parameters

-msDomainId
Specifies the ID of the managed domain to display. (String, required)

Return value

The command returns the ID for each domain version that exists for the managed set domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllMSDomainVersionIds('[-msDomainId ms1:domain1]')
```

- Using Jython list:

```
AdminTask.dpGetAllMSDomainVersionIds(['-msDomainId', 'ms1:domain1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllMSDomainVersionIds('-interactive')
```

dpGetMSDomain

The dpGetMSDomain command displays a domain in a DataPower appliance manager managed set.

Target object

None.

Required parameters

-msDomainId
Specifies the ID of the managed domain to display. (String, required)

Return value

The command returns a properties object that contains the attributes for the managed set domain of interest. The following table provides additional information about each attribute that the command returns:

Attribute	Description
msDomainId	Displays the ID of the domain.
managedSetId	Displays the ID of the managed set to which the domain belongs.
name	Displays the name of the domain.
desiredDomainVersionId	Displays the ID of the domain version that the managed set uses for the domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetMSDomain('[-msDomainId ms1:domain1]')
```

- Using Jython list:

```
AdminTask.dpGetMSDomain(['-msDomainId', 'ms1:domain1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetMSDomain('-interactive')
```

dpGetMSDomainVersion

The dpGetMSDomainVersion command displays a DataPower appliance manager managed domain version.

Target object

None.

Required parameters

-msDomainVersionId
Specifies the ID of the managed domain version of interest. (String, required)

Return value

The command returns a properties object that contains the attributes of the managed set domain version of interest. The following table provides additional information about the attributes that the command returns:

Attribute	Description
msDomainVersionId	Displays the ID of the managed domain version.
msDomainId	Displays the ID of the managed domain.
versionNumber	Displays the version number.
timestamp	Displays the date that the system created the copy.
userComment	Displays the comment that is stored with the managed set domain version.
isInUse	Displays whether the managed set uses the domain version for the domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetMSDomainVersion('[-msDomainVersionId ms1:domain1:1]')
```

- Using Jython list:

```
AdminTask.dpGetMSDomainVersion(['-msDomainVersionId', 'ms1:domain1:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetMSDomainVersion('-interactive')
```

dpManageDomain

The dpManageDomain command adds the domain to a managed set, and starts managing the domain.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set to manage. (String, required)

-domain
Specifies the name of the domain on the master appliance to manage. (String, required)

Return value

The command returns the ID of the task that the command creates. When the task completes, the value of the result attribute contains the ID of the new domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpManageDomain('[-managedSetId testMS1 -domain default]')
```

- Using Jython list:

```
AdminTask.dpManageDomain(['-managedSetId', 'testMS1', '-domain', 'default'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpManageDomain('-interactive')
```

dpRemoveMSDomainVersion

The dpRemoveMSDomainVersion command removes a managed domain version from the DataPower appliance manager.

Target object

None.

Required parameters

-msDomainVersionId
Specifies the ID of the managed domain version to remove. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpRemoveMSDomainVersion('[-msDomainVersionId ms1:domain1:1]')
```

- Using Jython list:

```
AdminTask.dpRemoveMSDomainVersion(['-msDomainVersionId', 'ms1:domain1:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpRemoveMSDomainVersion('-interactive')
```

dpSetMSDomain

The dpSetMSDomain command modifies a DataPower appliance manager managed domain.

Target object

None.

Required parameters

-msDomainId
Specifies the ID of the managed domain to modify. (String, required)

Optional parameters

-desiredDomainVersionId
Specifies the ID of the domain version to synchronize to each appliance in the managed set. (String, optional)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetMSDomain('[-msDomainId  ms1:domain1 -desiredDomainVersionId ms1:domain1:1]')
```

- Using Jython list:

```
AdminTask.dpSetMSDomain(['-msDomainId', 'ms1:domain1', '-desiredDomainVersionId', 'ms1:domain1:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetMSDomain('-interactive')
```

dpSetMSDomainVersion

The dpSetMSDomainVersion command modifies a DataPower appliance manager managed domain version.

Target object

None.

Required parameters

-msDomainVersionId
Specifies the ID of the managed domain version to modify. (String, required)

Optional parameters

-userComment
Specifies the comment to store in the domain version. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetMSDomainVersion('[-msDomainVersionId ms1:domain1:2 -userComment "New Web
 Service proxy for banking application"]')
```

- Using Jython list:

```
AdminTask.dpSetMSDomainVersion(['-msDomainVersionId', 'ms1:domain1:2',  '-userComment',
 'New Web Service proxy for banking application'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetMSDomainVersion('-interactive')
```

dpUnmanageDomain

The dpUnmanageDomain command removes the domain from a managed set, and stops managing the domain.

Target object

None.

Required parameters

-msDomainId
Specifies the ID of the managed domain to remove from the managed set. (String, required)

Optional parameters

-clean
Deletes the domain from each appliance in the managed set. (String, optional)

Return value

If you specify the clean parameter, the command returns the ID of the task that the system creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpUnmanageDomain('-msDomainId testMS1:default -clean')
```

- Using Jython list:

```
AdminTask.dpUnmanageDomain(['-msDomainId', 'testMS1:default', '-clean'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpUnmanageDomain('-interactive')
```

dpCopyMSSettingsVersion

The dpCopyMSSettingsVersion command copies a DataPower appliance manager managed settings version to a new managed set.

Target object

None.

Required parameters

-managedSetId
Specifies the ID of the managed set of interest. (String, required)

-msSettingsVersionId
Specifies the ID of the managed settings version to copy. (String, required)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpCopyMSSettingsVersion('[-managedSetId myManagedSet -msSettingsVersionId "myManagedSet1:1"]')
```

- Using Jython list:

```
AdminTask.dpCopyMSSettingsVersion(['-managedSetId', 'myManagedSet', ,-msSettingsVersionId',
 'myManagedSet1:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpCopyMSSettingsVersion('-interactive')
```

dpGetAllMSSettingsVersionIds

The dpGetAllMSSettingsVersionIds command displays the IDs of each settings version in a DataPower appliance manager managed set.

Target object

None.

Required parameters

-msSettingsId
Specifies the ID of the managed settings of interest. (String, required)

Return value

The command returns a string array that contains the IDs of the versions of the managed set settings of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetAllMSSettingsVersionIds('-msSettingsId ms1:2')
```

- Using Jython list:

```
AdminTask.dpGetAllMSSettingsVersionIds(['-msSettingsId', 'ms1:2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetAllMSSettingsVersionIds('-interactive')
```

dpGetMSSettings

The dpGetMSSettings command displays the ID of the managed set to which the settings belong.

Target object

None.

Required parameters

-msSettingsId
Specifies the ID of the managed settings of interest. (String, required)

Return value

The command returns a properties object that contains the attributes of the settings of interest. The following table provides additional information about the attributes that the command returns:

Attribute	Description
msSettingsId	Displays the ID of the settings.
managedSetId	Displays the ID of the managed ID to which the settings are assigned.
desiredSettingsVersionId	Displays the ID of the settings version that the managed set uses.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetMSSettings('-msSettingsId ms1:2')
```

- Using Jython list:

```
AdminTask.dpGetMSSettings(['-msSettingsId', 'ms1:2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetMSSettings('-interactive')
```

dpGetMSSettingsVersion

The dpGetMSSettingsVersion command displays a DataPower appliance manager managed settings version.

Target object

None.

Required parameters

-msSettingsVersionId
Specifies the ID of the managed settings version of interest. (String, required)

Return value

The command returns a properties object that contains the attributes of the managed set settings version of interest. The following table provides additional information about the attributes that the command returns:

Attribute	Description
msSetSettingsVersionId	Displays the ID of the managed set settings version.
msSettingsId	Displays the ID of the managed set settings of interest.
timestamp	Displays the date that the system created the copy.
versionNumber	Displays the number of the version.
userComment	Displays the comment associated with the managed set settings version.
isInUse	Displays the ID of the settings version that the managed set uses.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetMSSettingsVersion('-msSettingsVersionId ms1:2')
```

- Using Jython list:

```
AdminTask.dpGetMSSettingsVersion(['-msSettingsVersionId', 'ms1:2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetMSSettingsVersion('-interactive')
```

dpRemoveMSSettingsVersion

The dpRemoveMSSettingsVersion command removes a managed settings version from the DataPower appliance manager.

Target object

None.

Required parameters

-msSettingsVersionId
Specifies the ID of the managed settings version to remove. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpRemoveMSSettingsVersion('-msSettingsVersionId ms1:2')
```

- Using Jython list:

```
AdminTask.dpRemoveMSSettingsVersion(['-msSettingsVersionId', 'ms1:2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpRemoveMSSettingsVersion('-interactive')
```

dpSetMSSettings

The dpSetMSSettings command modifies the DataPower appliance manager managed settings.

Target object

None.

Required parameters

-msSettingsId
Specifies the ID of the managed settings to modify. (String, required)

Optional parameters

-desiredSettingsVersionId
Specifies the ID of the settings version to synchronize to each appliance in the managed set. (String, optional)

Return value

The command returns the ID of the task that the command creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetMSSettings('-msSettingsId ms1 -desiredSettingsVersionId ms1:1')
```

- Using Jython list:

```
AdminTask.dpSetMSSettings(['-msSettingsId', 'ms1', '-desiredSettingsVersionId', 'ms1:1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetMSSettings('-interactive')
```

dpSetMSSettingsVersion

The dpSetMSSettingsVersion command modifies a DataPower appliance manager managed settings version.

Target object

None.

Required parameters

-msSettingsVersionId
Specifies the ID of the managed settings version to modify. (String, required)

Optional parameters

-userComment
Specifies the comment to store in the settings version. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpSetMSSettingsVersion('[-msSettingsVersionId ms1:2 -userComment "has new
 timeserver added"]')
```

- Using Jython list:

```
AdminTask.dpSetMSSettingsVersion(['-msSettingsVersionId', 'ms1:2', '-userComment', 'has
 new timeserver added'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpSetMSSettingsVersion('-interactive')
```

dpGetAllTaskIds

The dpGetAllTaskIds command displays the IDs of each of the DataPower appliance manager tasks.

Target object

None.

Return value

The command returns a list of each asynchronous task ID that currently exists in the DataPower applance manager.

Batch mode example usage

- Using Jython string and list:

```
AdminTask.dpGetAllTaskIds()
```

dpGetTask

The dpGetTask command displays information for a specific DataPower appliance manager task.

Target object

None.

Required parameters

-taskId
Specifies the ID of the task of interest. (String, required)

Return value

The command returns information about the asynchronous task of interest, including the following attributes:

Attribute	Description
taskId	Displays the unique ID of the task.
taskDescription	Displays a description of the task.
creationDate	Displays the date and time that the system created the task.
createdByUser	Displays the user that created the task.
currentStep	Displays the current step number for the task.
currentStepDescription	Displays a description of the current step.
currentStepTimestamp	Displays the time and date that the current step was last updated in the task. The system updates the currentTimeStamp attribute when the system constructs the object, updates a step, marks the task complete, or experiences an exception.
error	Displays the error message of the exception that caused the task to end, if applicable.
totalSteps	Displays the estimated total number of steps for the task. Do not use this argument to determine if the task is complete. Refer to the isComplete attribute to determine if the task is complete.
hasError	Displays a value of `true` if the task is not successfully completed.
hasUpdate	Displays a value of `true` if the task is updated.
isComplete	Displays a value of `true` if the task is complete.
taskStatus	Displays an integer that represents the status of the task. If the returned value is `0`, then the task is in a queue and the system has not started the task. If the returned value is `1`, then the task is in progress. If the returned value is `2`, then the task completed successfully. If the returned value is `3`, then the task experienced an exception.
result	Displays the result that the task returns. Refer to the specific command to determine if the command returns output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpGetTask('-taskId 1')
```

- Using Jython list:

```
AdminTask.dpGetTask(['-taskId', '1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpGetTask('-interactive')
```

dpPurgeTask

The dpPurgeTask command purges a specific DataPower appliance manager task. The system automatically deletes tasks after 24 hours.

Target object

None.

Required parameters

-taskId
Specifies the ID of the task in the DataPower appliance manager to purge. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.dpPurgeTask('-taskId 1')
```

- Using Jython list:

```
AdminTask.dpPurgeTask(['-taskId', '1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.dpPurgeTask('-interactive')
```

Chapter 8. Managing servers and nodes with scripting

Use the wsadmin tool to administer your administrative architecture and runtime settings.

- Use the following topics to manage node configurations with the wsadmin tool:
  1. “Stopping a node using scripting” on page 464
  2. “Restarting node agent processes using the wsadmin tool” on page 465
- Use the following topics to manage server configurations with the wsadmin tool:
  1. “Starting servers using scripting” on page 465
  2. “Stopping servers using scripting” on page 466
  3. “Querying server state using scripting” on page 467
  4. “Listing running applications on running servers using scripting” on page 468
  5. “Starting listener ports using scripting” on page 470
  6. “Managing generic servers using scripting” on page 471
  7. “Setting development mode for server objects using scripting” on page 472
  8. “Disabling parallel startup using scripting” on page 473
  9. “Obtaining server version information with scripting” on page 473
- Use the following topics to view wsadmin command reference information for server and node management:
  1. “UnmanagedNodeCommands command group for the AdminTask object” on page 523
  2. “ManagedObjectMetadata command group for the AdminTask object” on page 490
  3. “Utility command group of the AdminTask object” on page 488
  4. “ConfigArchiveOperations command group for the AdminTask object” on page 526
- Use the following topics to use properties files to manage your environment:
  1. “Managing environment configurations using properties files” on page 447
  2. “Extracting properties files” on page 450
  3. “Validating properties files” on page 453
  4. “Applying properties files” on page 455
  5. “Creating server, cluster, application, or authorization group objects using properties files” on page 458
  6. “Deleting server, cluster, application, or authorization group objects using properties files” on page 460
  7. “Creating and deleting configuration objects using properties files” on page 462

Administering jobs in a flexible management environment using scripting

Use the flexible management environment to locally or remotely submit and manage administrative jobs. You can use the job manager to manage applications, modify configurations, and control the application server run time.

About this task

You can coordinate management actions among multiple deployment managers, asynchronously administer multiple unfederated application servers, and submit jobs to start servers from a management profile that contains a job manager. In order to begin using the job manager to run jobs, register your application server and deployment manager nodes as managed nodes of the job manager.

Use the following steps to use a flexible management environment:

- Create an administrative agent management node.
- Create a job manager management node.
- Register application server and deployment manager nodes as managed nodes on the job manager.
- Create managed node groups to submit administrative jobs across nodes.
- Submit administrative jobs.
- Monitor job progress.
- Manage your flexible management configuration.

Registering nodes with the job manager using scripting

Configure your flexible management environment by registering application servers that are registered with administrative agents, or by registering deployment managers as nodes on the job manager. After you register nodes with the job manager, you can submit and manage jobs.

Before you begin

You must configure a flexible management environment, consisting of a job manager, and optionally an administrative agent for the application server node to register. Start the job manager and the administrative agent processes before registering nodes or deployment managers with the job manager.

Note: The identity under which the administrative agent server is running must have at least the monitor role for the profile of the managed application server node. Otherwise, when you attempt to register the node with the job manager, registration fails with the ADMN0022E message.

About this task

You can administer multiple application servers that run customer applications from a management profile that contains an administrative agent. The administrative agent provides a single administrative console to administer the application servers.

You can coordinate management actions among multiple deployment managers, asynchronously administer multiple unfederated application servers, and submit jobs to start servers from a management profile that contains a job manager. In order to begin using the job manager to run jobs, register your application server and deployment manager nodes as managed nodes of the job manager.

Use the following steps to register profiles that contain administrative agents as nodes on the job manager.

1. Register the application server with the administrative agent if it is not yet registered.

   Run the registerNode command from the `bin` directory for the administrative agent server to register a node with the administrative agent. When you run the command, the standalone node is converted into a node that the administrative agent manages. The administrative agent and the node being registered must be on the same system. You can only run the command on an unfederated node. If the command is run on a federated node, the command exits with an error.

   If the administrative console or the management Enterprise JavaBeans (EJB) applications of the application server being registered are enabled, the node registration process disables them.

   Use the registerNode command utility to register the application server profile with the administrative agent, as the following command demonstrates:

```
bin>registerNode -profilePath profile_root/profiles/AppSrv01 -host localhost -conntype SOAP -port 8878
```

2. Launch the wsadmin tool. Navigate to the profile_root/profiles/*myAdminAgent*/bin directory and use the following command to connect the wsadmin tool to the administrative agent process:

```
wsadmin -profileName myAdminAgent -lang jython
```

3. Register the node as a node on the job manager.

   If the node to register contains an administrative agent, use the registerWithJobManager command and the following parameters to register a node as a node on the job manager.

Parameter	Description	Data type
-managedNodeName	Specifies the name of the node that is registered with the administrative agent. If the node is a deployment manager profile, specify the node name of the deployment manager. (Required)	String
-host	Optionally specifies the hostname of the job manager.	String
-port	Optionally specifies the administrative port number to use. The default secure port number is `9943`. The default unsecure port number is `9960`.	
-user	Optionally specifies the connector login user name.	String
-password	Optionally specifies the password for the connector login user name.	String
-alias	Optionally specifies an alias for the node. The job manager uses this name instead of the value of the managedNodeName parameter to register the node. Use this parameter if the new node has the same name of a node that is registered with the job manager.	String
-startPolling	Optionally specifies whether to start polling after registering the node. Specify `false` to disable polling. The default value is `true`.	Boolean
-autoAcceptSigner	Optionally specifies whether to automatically accept the signer provided by the server. Specify `false` to disable this option. The default value is `true`.	Boolean

The following sample command registers the `AppSvr01` application server profile with the job manager:

```
AdminTask.registerWithJobManager('[-host jobMgrHost -managedNodeName myhostNode01]')
```

The following sample command registers the `DMGR01` deployment manager profile with the job manager:

```
AdminTask.registerWithJobManager('[-host jobMgrHost -managedNodeName DMGR01]')
```

4. Optional: Repeat the registerWithJobManager command to register additional profiles as nodes on the job manager.

Results

The node of interest is registered with the job manager when the system successfully runs the registerWithJobManager command.

What to do next

Submit, monitor, and manage jobs for the nodes that are registered with the job manager.

Grouping nodes in a flexible management environment using scripting

Define groups of nodes to simplify large environment configurations. Node groups allow you to submit jobs to multiple nodes.

Before you begin

Create a job manager and register one or more nodes in a flexible management environment. Verify that the job manager process is running.

About this task

You can coordinate management actions among multiple deployment managers, asynchronously administer multiple unfederated application servers, and submit jobs to start servers from a management profile that contains a job manager. In order to begin using the job manager to run jobs, register your application server and deployment manager nodes as managed nodes of the job manager.

Node groups can be used to improve your system administration, including easy management of branch office systems and server farms. For example, a business has 1000 global store locations, and each location has three application servers. Each store location is connected to the business headquarters data center. The business manages all systems and runs jobs from the business headquarters. In this scenario, the business can easily manage the branch offices by creating one node group per continent.

1. Launch the wsadmin tool.

Navigate to the app_server_root/bin directory and run the following command to launch the wsadmin tool and connect to the job manager process:

```
wsadmin -profileName myJobManager -lang jython
```

2. Create a node group.

   Use the createManagedNodeGroup command to create a new node group in your flexible management configuration:

```
AdminTask.createManagedNodeGroup('-groupName European_Branch_Offices -description "Management group for all European branch offices"')
```

3. Add nodes to the node group.

   Use the addMemberToManagedNodeGroup command to add nodes to the new group. Each node can belong to one or more groups. The following example commands add the Hursley_office, Berlin_office, Warsaw_office, and Paris_office nodes to the European_Branch_Offices node group:

```
AdminTask.addMemberToManagedNodeGroup('-groupName European_Branch_Offices -managedNodeNameList "[Hursley_Node01][Berlin_Node01][Warsaw_Node01][Paris_Node01]"')
```

4. Optional: Verify that the system added the nodes to the node group.

   Use the getManagedNodeGroupMembers command to display the nodes for the node group of interest, as the following example displays:

```
AdminTask.getManagedNodeGroupMemebers('-groupName European_Branch_Offices')
```

   The command returns a list of node names that are registered with the node group of interest, as the following sample output displays:

```
"[Hursley_Node01][Berlin_Node01][Warsaw_Node01][Paris_Node01]"
```

Results

A node group with node group members exists in your flexible management configuration.

What to do next

Use the job manager to submit, monitor, and manage jobs for your node groups.

Running administrative jobs using scripting

Use this topic to submit and track administrative jobs in a flexible management environment using the wsadmin tool.

Before you begin

Configure a job manager, administrative agent, and register nodes and deployment managers with the job manager to set up a flexible management environment. You can optionally create management groups to simultaneously submit a job to multiple nodes.

About this task

After configuring your flexible management environment, you can submit, monitor, and manage jobs for the nodes that are registered with the job manager.

1. Launch the wsadmin tool. Navigate to the app_server_root/bin directory and use the following command to connect the wsadmin tool to the job manager process:

```
wsadmin -profileName myJobManager -lang jython
```

2. Submit the administrative job to the job manager.

   Use the submitJob command to submit administrative jobs. Job submissions consist of the following information:

   Job type

   The job type specifies the type of job to perform. Many jobs exist in the flexible management environment including application management, configuration, and application server runtime control jobs.

Job target list and target group

The job target list and group specifies the nodes and node groups where the job runs.

Job specific parameters

Most administrative jobs require information in addition to the job type and target in order to run the job. Job parameters are specific to each job type.

Optional generic parameters

In addition to the job specific parameters, you can include any of the following optional parameters with the job submission:

Parameter	Description	Type
username	Specifies the username to use to submit the job when security is enabled.	String
password	Specifies the password for the username to use to submit the job when security is enabled.	String
description	Specifies a description for the job.	String
activationDateTime	Specifies the date and time to activate the job in the format `"2006-05-03T10:30:45-0000"`. The `"-0000"` section of the activationDateTime parameter value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `"2006-05-03T10:30:45Z"`. If you do not specify the time zone, the system uses the time zone of the server.	String
expirationDateTime	Specifies the expiration date for the job, in the format `"2006-05-03T10:30:45-0000"`. The `"-0000"` section of the -activationDateTime parameter value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `"2006-05-03T10:30:45Z"`. If you do not specify the time zone, the system uses the time zone of the server.	String
executionWindow	Specifies the recurring interval for the job.	String
executionWindowUnit	Specifies the recurring interval unit of measure for the value set by the executionWindow parameter. Specify `DAILY` to run the job daily, `WEEKLY` to run the job weekly, `MONTHLY` to run the job monthly, `YEARLY` to run the job annually, or `CONNECTION` to run the job each time the node connects and polls for jobs.	String
email	Specifies the email address that the system sends job notifications to.	String

The following example submits a job to start an application server. The following command example submits the start application job, and sets the returned job token to the `MyStartJob` variable:

```
myStartJob = AdminTask.submitJob('-jobType startApplication -targetList "[MyNode01]" -jobParams "[applicationName myApplication]" -email admin@company.com')
```

3. Optional: Monitor the job status.

   Use the getOverallJobStatus command to display the status of the job, as the following example displays:

```
AdminTask.getOverallJobStatus('[-jobTokenList [myStartJob]]')
```

   If you did not set the myStartJob variable in the previous step, specify the return value from the **submitJob** command for the -jobTokenList parameter.

   The command returns job status information for the job or jobs of interest. The system displays the following information in the overall job status:

   - The STATE attribute specifies the current state of the job.
   - The TOTAL_RESULTS attribute specifies the total number of jobs.
   - The DISTRIBUTED attribute specifies the number of distributed jobs.
   - The ASYNC_IN_PROGRESS attribute specifies the number of asynchronous jobs in progress.
   - The SUCCEEDED attribute specifies the number of successful jobs.
   - The FAILED attribute specifies the number of failed jobs.
   - The DELAYED attribute specifies the number of delayed jobs.
   - The REJECTED attribute specifies the number of rejected jobs.
   - The NOT_ATTEMPTED attribute specifies the number of jobs that the system has not attempted.

What to do next

Submit additional administrative jobs to the job manager and monitor existing jobs. You can also schedule future administrative jobs.

Running administrative jobs across multiple nodes using scripting

Use this topic to run administrative jobs across multiple nodes in a flexible management environment using the wsadmin tool.

Before you begin

Configure a job manager, administrative agent, and register nodes with the job manager to set up a flexible management environment.

About this task

After configuring your flexible management environment, you can submit, monitor, and manage jobs for the nodes that are registered with the job manager.

In the following example, a company has four branch offices located in Hursley, Berlin, Warsaw, and Paris, with an application server at each location. An application that runs on each application server frequently experiences a memory leak issue. While a development team fixes the application, it might be necessary for each location to frequently stop and restart the application server. Use the following steps to create a node group consisting of the application servers from each location, and schedule weekly recurring jobs to stop and restart the four application servers.

1. Launch the wsadmin tool. Navigate to the app_server_root/bin directory and use the following command to connect the wsadmin tool to the job manager process:

```
wsadmin -profileName myJobManager -lang jython
```

2. Create a node group.

   Use the createManagedNodeGroup command to create a new node group in your flexible management configuration:

```
AdminTask.createManagedNodeGroup('-groupName European_Branch_Offices -description
 "Management group for all European branch offices"')
```

3. Add nodes to the node group.

   Use the addMemberToManagedNodeGroup command to add nodes to the new group. Each node can belong to one or more groups. The following example commands add the `Hursley_office`, `Berlin_office`, `Warsaw_office`, and `Paris_office` nodes to the `European_Branch_Offices` node group:

```
AdminTask.addMemberToManagedNodeGroup('-groupName European_Branch_Offices
 -managedNodeNameList "[Hursley_Node01][Berlin_Node01][Warsaw_Node01][Paris_Node01]"')
```

4. Schedule an administrative job on the job manager.

   Use the submitJob command to submit the future administrative job. Job submissions consist of the following information:

   Job type
   The job type specifies the type of job to perform. Many jobs exist in the flexible management environment including application management, product maintenance, configuration, and application server runtime control jobs.

   Job target list and target group
   The job target list and target group specifies the nodes and node group where the job runs.

   Job specific parameters
   Most administrative jobs require configuration information in addition to the job type and target in order to run the job. Job parameters are specific to each job type.

Optional generic parameters

In addition to the job specific parameters, you can include any of the following optional parameters with the job submission:

Parameter	Description	Type
username	Specifies the username to use to submit the job when security is enabled.	String
password	Specifies the password for the username to use to submit the job when security is enabled.	String
description	Specifies a description for the job.	String
activationDateTime	Specifies the date and time to activate the job in the format `"2006-05-03T10:30:45-0000"`. The `-0000` section of the activationDateTime parameter value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `"2006-05-03T10:30:45Z"`. If you do not specify the time zone, the system uses the time zone of the server.	String
expirationDateTime	Specifies the expiration date for the job, in the format `"2006-05-03T10:30:45-0000"`. The `-0000` section of the activationDateTime parameter value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `"2006-05-03T10:30:45Z"`. If you do not specify the time zone, the system uses the time zone of the server.	String
expireAfter	Specifies the amount of time, in minutes, to wait before the job expires.	Integer
executionWindow	Specifies the recurring interval for the job.	String
executionWindowUnit	Specifies the recurring interval unit of measure for the value set by the executionWindow parameter. Specify `DAILY` to run the job daily, `WEEKLY` to run the job weekly, `MONTHLY` to run the job monthly, `YEARLY` to run the job annually, or `CONNECTION` to run the job each time the node connects and polls for jobs.	String
email	Specifies the email address where the system sends job notifications.	String

The following example schedules two weekly recurring jobs. For this example, an application that runs on application servers at 4 branch offices frequently experiences a memory leak issue. While a development team fixes the application, it might be necessary to frequently stop and restart the application server. The following command examples schedule the job manager to stop and restart the server once per week, and notifies the system administrator when the server stops and restarts:

```
AdminTask.submitJob('-jobType stopServer -group European_Branch_Offices
 -jobParams "[serverName server1]" -activationDateTime "2006-05-03T10:30:45Z" -executionWindowUnit DAILY
 -executionWindow 13:00:00-14:00:00 -email system_admin@company.com')  AdminTask.submitJob('-jobType
 startServer -group European_Branch_Offices -jobParams "[serverName server1]" -activationDateTime
 "2006-05-03T10:40:45Z" -recurringIntervalUnits DAILY -recurringInterval 13:00:00-14:00:00 -email
 system_admin@company.com')
```

Results

The jobs have been submitted to the queue and will run at the date and time specified by the command.

What to do next

Use the commands in the AdministrativeJobs command group to manage and query for administrative jobs in your flexible management configuration.

Scheduling future administrative jobs using scripting

Use this topic to schedule future recurring administrative jobs in a flexible management environment using the wsadmin tool.

Before you begin

Configure a job manager, administrative agent, and register nodes with the job manager to set up a flexible management environment. You can optionally create management groups to simultaneously submit a job to multiple nodes.

About this task

After configuring your flexible management environment, you can submit, monitor, and manage jobs for the nodes that are registered with the job manager.

1. Launch the wsadmin tool. Navigate to the app_server_root/bin directory and use the following command to connect the wsadmin tool to the job manager process:

```
wsadmin -profileName myJobManager -lang jython
```

2. Schedule a future administrative job to the job manager.

   Use the **submitJob** command to submit the future administrative job. Job submissions consist of the following information:

   Job type
   The job type specifies the type of job to perform. Many jobs exist in the flexible management environment including application management, product maintenance, configuration, and application server runtime control jobs.

   Job target
   The job target specifies the node where the job runs.

   Job specific parameters
   Most administrative jobs require configuration information in addition to the job type and target in order to run the job. Job parameters are specific to each job type.

   Optional generic parameters
   In addition to the job specific parameters, you can include any of the following optional parameters with the job submission:

Parameter	Description	Type
username	Specifies the username to use to submit the job when security is enabled.	String
password	Specifies the password for the username to use to submit the job when security is enabled.	String
description	Specifies a description for the job.	String
activationDateTime	Specifies the date and time to activate the job in the format "2006-05-03T10:30:45-0000". The -0000 section of the activationDateTime parameter value represents RFC 822 format. You can specify Z as a shortcut for Greenwich Mean Time (GMT), such as "2006-05-03T10:30:45Z". If you do not specify the time zone, the system uses the time zone of the server.	String
expirationDateTime	Specifies the expiration date for the job, in the format "2006-05-03T10:30:45-0000". The -0000 section of the activationDateTime parameter value represents RFC 822 format. You can specify Z as a shortcut for Greenwich Mean Time (GMT), such as "2006-05-03T10:30:45Z". If you do not specify the time zone, the system uses the time zone of the server.	String
expireAfter	Specifies the amount of time, in minutes, to wait before the job expires.	Integer
executionWindow	Specifies the recurring interval for the job.	String
executionWindowUnit	Specifies the recurring interval unit of measure for the value set by the executionWindow parameter. Specify DAILY to run the job daily, WEEKLY to run the job weekly, MONTHLY to run the job monthly, YEARLY to run the job annually, or CONNECTION to run the job each time the node connects and polls for jobs.	String
email	Specifies the email address that the system sends job notifications to.	String

   The following example schedules two weekly recurring jobs. For this example, an application frequently experiences a memory leak issue. While a development team fixes the application, it might be necessary to frequently stop and restart the application. The following command examples schedule the job manager to stop and restart the server once per week, and notifies the system administrator when the server stops and restarts:

```
AdminTask.submitJob('-jobType stopServer -targetList "[MyNode01]" -jobParams
 "[serverName server1]" -activationDateTime "2006-05-03T10:30:45Z" -executionWindowUnit DAILY
 -executionWindow 13:00:00-14:00:00 -email system_admin@company.com')  AdminTask.submitJob('-jobType
 startServer -targetList "[MyNode01]" -jobParams "[serverName server1]
 " -activationDateTime "2006-05-03T10:40:45Z" -executionWindowUnit DAILY -executionWindow
 13:00:00-14:00:00 -email system_admin@company.com')
```

Results

The jobs have been submitted to the queue and will run at the date and time specified by the command.

What to do next

Submit additional administrative jobs to the job manager and monitor existing jobs.

Managing administrative jobs using scripting

Use the wsadmin tool and the commands in the AdministrativeJobs command group to manage administrative jobs in your flexible management environment.

Before you begin

Configure a job manager, administrative agent, and register managed nodes with the job manager to set up a flexible management environment.

About this task

Use this topic to manage administrative jobs that you submit to the job manager.

- Display the status of a job.

  Use the getOverallJobStatus command to display the overall job status for a specific job or a list of jobs of interest. The following command example displays the job status for a specific job:

```
AdminTask.getOverallJobStatus(['-jobTokenList "[myJobToken]"'])
```

  The following command example displays the overall job status for multiple jobs:

```
AdminTask.getOverallJobStatus('-jobTokenList "[myJobToken, myJobToken2, myJobToken3]"')
```

  The command returns job status information for the job or jobs of interest. The system displays the following information in the overall job status:

  - The STATE attribute specifies the current state of the job.
  - The TOTAL_RESULTS attribute specifies the total number of jobs.
  - The DISTRIBUTED attribute specifies the number of distributed jobs.
  - The ASYNC_IN_PROGRESS attribute specifies the number of asynchronous jobs in progress.
  - The SUCCEEDED attribute specifies the number of successful jobs.
  - The PARTIALLY_SUCCEEDED attribute specifies the number of partially successful jobs. Partial success can occur, for example, when a node represents multiple servers, and only some of the servers on the node complete successfully.
  - The FAILED attribute specifies the number of failed jobs.
  - The REJECTED attribute specifies the number of rejected jobs.
  - The NOT_ATTEMPTED attribute specifies the number of jobs that the system has not attempted.

- Suspend a job.

  Use the suspendJob command to suspend a job on the job manager, as the following command demonstrates:

```
AdminTask.suspendJob('-jobToken myToken')
```

- Resume a job.

  Use the resumeJob command to resume a suspended job, as the following command demonstrates:

```
AdminTask.resumeJob('-jobToken myToken')
```

- Delete a job.

  Use the deleteJob command to delete an existing job from the job manager. If the job is running when you invoke the command, the system still returns the job results regardless of whether the job is deleted. The following command example deletes a job from the job manager:

```
AdminTask.deleteJob('-jobToken myToken -deleteResults true')
```

Administrative job types

In a flexible management environment, you can use the wsadmin tool to submit administrative jobs to the job manager. This topic provides detailed information about the administrative jobs, the job parameters, and sample command syntax.

In a flexible management environment, you can configure an administrative agent and job manager to submit jobs to multiple nodes or node groups in your configuration. Then, you can submit administrative jobs to queue jobs across your managed environment. Each administrative job has a corresponding job type, which defines the required parameters to submit the job. You can use the commands in the AdministrativeJobs command group to submit administrative jobs to the job manager.

You can submit administrative jobs to manage your applications.

Display parameters for all jobs

Use the submitJob command to submit administrative jobs. Job submissions consist of the following information:

Job type
The job type specifies the type of job to perform. Many jobs exist in the flexible management environment including application management, product maintenance, configuration, and application server runtime control jobs.

Job target
The job target specifies the managed node where the job runs.

Job specific parameters
Most administrative jobs require configuration information in addition to the job type and target in order to run the job. Job parameters are specific to each job type.

Optional generic parameters
In addition to the job specific parameters, you can include any of the following optional parameters with the job submission:

Parameter	Description	Type
username	Specifies the username to use to submit the job when security is enabled.	String
password	Specifies the password for the username to use to submit the job when security is enabled.	String
description	Specifies a description for the job.	String
activationDateTime	Specifies the date and time to activate the job in the format "2006-05-03T10:30:45-0000". The -0000 section of the activationDateTime parameter value represents RFC 822 format. You can specify Z as a shortcut for Greenwich Mean Time (GMT), such as "2006-05-03T10:30:45Z". If you do not specify the time zone, the system uses the time zone of the job manager.	String
expirationDateTime	Specifies the expiration date for the job, in the format "2006-05-03T10:30:45-0000". The -0000 section of the activationDateTime parameter value represents RFC 822 format. You can specify "Z" as a shortcut for Greenwich Mean Time (GMT), such as "2006-05-03T10:30:45Z". If you do not specify the time zone, the system uses the time zone of the job manager.	String
executionWindow	Specifies the recurring interval, if any, for the job.	String
executionWindowUnit	Specifies the unit of measure for the recurring interval. Valid values include HOURS, DAYS, WEEKS, MONTHS, CONNECT, and ONCE. The default value is HOURS.	String
email	Specifies the email address where the system sends job notifications.	String

Example usage

```
AdminTask.submitJob('-jobType installApplication -targetList
 [Node1,Node2,Node3] -jobParams "[applicationName myApplication]" -email admin@company.com')
```

Display supported jobs

The **inventory** administrative job displays a list of administrative jobs on the job manager queue for the job target of interest.

Job parameters

None.

Example usage

```
AdminTask.submitJob('-jobType inventory -targetList
 [Node1,Node2,Node3]')
```

Display job statuses

The **status** administrative job updates the managed node cache for each managed node that contains the `status` attribute.

The status administrative job returns job information for each of the following statuses:

- The `DISTRIBUTED` status specifies the number of incomplete jobs received by the agent for the node.
- The `SUCCEEDED` status specifies the number of jobs completed successfully.
- The `PARTIALLY_SUCCEEDED` status specifies the number of jobs partially completed. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The `FAILED` status specifies the number of jobs that failed.
- The `NOT_ATTEMPTED` status specifies the number of jobs that the system did not attempt.

Job parameters

None.

Example usage

```
AdminTask.submitJob('-jobType status -targetList
 [Node1,Node2,Node3]')
```

Submit a job that runs a wsadmin script

You can submit a job to run a wsadmin script on a managed node of the job manager. The managed node can be a deployment manager or an unfederated application server. Before running a wsadmin script on a managed node, transfer the wsadmin script from the job manager to the managed node using the distributeFile job. Then use the following job to run the wsadmin script.

Job parameters

Parameter	Description	Type
scriptFileLocation	Specifies the location of the script file to run on the managed node.	String

Example usage

```
AdminTask.submitJob('[-jobType runWsadminScript -targetList
 [DmgrManagedNodeName] -jobParams "[[scriptFileLocation jythonTestScript.py]]"]')
```

Submit jobs to manage applications

You can use the application management jobs to distribute, install, update, remove, and control applications on the application server. For example, you can submit the distributeFile, installApplication, and startApplication jobs to deploy applications in your environment. To remove applications from your environment, submit the stopApplication, uninstallApplication, and removeFile jobs. Use the following application management jobs to administer your application configurations:

collectFile

The collectFile administrative job collects a target ZIP file and transfers it to the job manager. If the source location is a directory instead of a file, the job recursively zips the directory contents and transfers the resulting ZIP file.

Job parameters

Parameter	Description	Type
source	Specifies the source location of the ZIP file or directory of interest. The system determines the file location is relative to the profile_root directory of the target node.	String
destination	Specifies the destination location. The default value is a directory named profile_root/config/temp/*JobManagerName*/ *jobToken*/*nodeName*.	String
distributionProvider	Optionally specifies the name of the distribution provider.	String

Example usage

The following example runs the collectFile job:

```
AdminTask.submitJob('[-jobType collectFile -targetList [Node1] -jobParams "[[source logs.zip][destination targetLocationOfFile]]"]')
```

distributeFile

The distributeFile administrative job transfers a package of deployable content from the job manager to the target nodes for the job. The system stores the application, or subset of application content, in a user-specified location at the target node until additional jobs are submitted to process the content. The installApplication and startApplication administrative jobs reference the destination that you provide to the distributeFile job as the identifier of the application content in the temporary location on the target node. Additionally, you can use this job to distribute script files and properties-based configuration files.

The file to distribute from the job manager initially must be in the /config/temp/JobManager directory of the job manager profile. Then, the system moves the file into the downloadedContent folder of the administrative agent profile.

Job parameters

Parameter	Description	Type
source	Specifies the source location of the content to distribute.	String
destination	Specifies the destination location on the target node where the system saves the content.	String
distributionProvider	Optionally specifies the name of the distribution provider.	String

Example usage

```
AdminTask.submitJob('-jobType distributeFile -targetList [Node1,Node2,Node3]
 -jobParams "[[source C:\applications\myApplication][destination C:\applicationsToInstall]]"')
```

removeFile

The removeFile administrative job removes each file system artifact for the content of interest from the target node. If you are removing all file system artifacts for an application, you must stop and uninstall the application before removing the files. The removeFile job only removes files that were created with the distributeFile job.

Job parameters

Parameter	Description	Type
location	Specifies the location of the file to remove from the target node.	String
distributionProvider	Optionally specifies the name of the distribution provider.	String

Example usage

```
AdminTask.submitJob('-jobType removeFile -targetList [Node1,Node2,Node3]
 -jobParams "[location applicationsToInstall]"')
```

startApplication

The startApplication administrative job starts a previously installed application on the target node. The system changes the application status to active, loads the application in the runtime, and opens the application to receive client requests. This is the last step in the application deployment process.

The startApplication administrative job logs one of the following results when it completes:

- The `DISTRIBUTED` value specifies that the agent for the node received the job, but the job is not complete.
- The `SUCCEEDED` value specifies that the job completed successfully.
- The `PARTIALLY_SUCCEEDED` value specifies that the job was partially completed. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The `FAILED` value specifies that the job failed.
- The `NOT_ATTEMPTED` value specifies that the system did not attempt to complete the job.

Job parameters

Parameter	Description	Type
applicationName	Specifies the name of the application to start.	String

Example usage

```
AdminTask.submitJob('-jobType startApplication -targetList [Node1,Node2,Node3]
 -jobParams "[applicationName myApplication]"')
```

stopApplication

The stopApplication administrative job stops the application on the target node. The system changes the application status to stopped and no longer receives client requests. You can use the startApplication job to restart the application. This is the first step in the application removal process.

The stopApplication administrative job logs one of the following results when it completes:

- The `DISTRIBUTED` value specifies that the agent for the node received the job, but the job is not complete.
- The `SUCCEEDED` value specifies that the job completed successfully.
- The `PARTIALLY_SUCCEEDED` value specifies that the job was partially completed. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The `FAILED` value specifies that the job failed.
- The `NOT_ATTEMPTED` value specifies that the system did not attempt to complete the job.

Parameter	Description	Type
applicationName	Specifies the name of the application to stop.	String

Example usage

```
AdminTask.submitJob('-jobType stopApplication -targetList [Node1,Node2,Node3]
 -jobParams "[applicationName myApplication]"')
```

installApplication

The installApplication administrative job installs and binds an application or application element into the target environment. You must distribute the deployable content to the targets before you can install the application.

Job parameters

Parameter	Description	Type
applicationName	Specifies an application name to use to identify the application to install.	String
appLocation	Optionally specifies the location of the application file to install. By default, the submitJob command appends the `.ear` file format notation to the application name, and searches for the application in the default destination location.	String
serverName	Optionally specifies the name of the server where the system installs the application.	String
nodeName	Optionally specifies the node of interest, which identifies the server in a Network Deployment cell.	String

Example usage

```
AdminTask.submitJob('-jobType installApplication -targetList
 [Node1,Node2,Node3] -jobParams "[[applicationName myApplication][appLocation applicationsToInstall]]"')
```

updateApplication

The updateApplication administrative job updates an application or application element in the target environment. You must distribute the deployable content to the targets before you can update the application.

Job parameters

Parameter	Description	Type
applicationName	Specifies an application name to use to identify the application to update.	String
appLocation	Optionally specifies the location of the application file to install. By default, the submitJob command appends the `.ear` file format notation to the application name, and searches for the application in the default destination location.	String
serverName	Optionally specifies the name of the server where the system updates the application.	String
nodeName	Optionally specifies the node of interest, which identifies the server in a Network Deployment cell.	String

Example usage

```
AdminTask.submitJob('-jobType updateApplication -targetList [Node1,Node2,Node3]
 -jobParams "[[applicationName myApplication][appLocation applicationsToUpdate]]"')
```

uninstallApplication

The uninstallApplication administrative job removes the application from the application server after you stop the application. Use the application removeFile job to remove all file system artifacts for the application of interest from the target node.

Job parameters

Parameter	Description	Type
applicationName	Specifies the name of the application to remove.	String

Example usage

```
AdminTask.submitJob('-jobType uninstallApplication -targetList
 [Node1,Node2,Node3] -jobParams "[applicationName myApplication]"')
```

Submit jobs to manage servers

The system administrator can use the application server configuration jobs to modify the configuration for remote and local application servers.

createApplicationServer

The createApplicationServer administrative job creates an application server in the target environment.

Job parameters

Parameter	Description	Type
serverName	Specifies the name of the application server to create.	String
nodeName	Optionally specifies the node of interest, which identifies the server in a Network Deployment cell.	String

Example usage

```
AdminTask.submitJob('-jobType createApplicationServer -targetList
 [Node1,Node2,Node3] -jobParams "[serverName AppServer01]"')
```

deleteApplicationServer

The deleteApplicationServer administrative job removes an application server from the target environment.

Job parameters

Parameter	Description	Type
serverName	Specifies the name of the application server to delete.	String
nodeName	Optionally specifies the node of interest, which identifies the server in a Network Deployment cell.	String

Example usage

```
AdminTask.submitJob('-jobType deleteApplicationServer -targetList
 [Node1,Node2,Node3] -jobParams "[serverName AppServer01]"')
```

createProxyServer

The createProxyServer administrative job creates a proxy server in a cell of a deployment manager that is a managed node of a job manager.

Job parameters

Parameter	Description	Type
serverName	Specifies the name of the proxy server to create.	String
nodeName	Optionally specifies the node in which the proxy server will reside.	String

Example usage

```
AdminTask.submitJob('[-jobType createProxyServer -jobParams "[[serverName
 testServer1][nodeName testNode1]]" ]')
```

deleteProxyServer

The deleteProxyServer deletes a proxy server from a cell of a deployment manager that is a managed node of a job manager.

Job parameters

Parameter	Description	Type
serverName	Specifies the name of the proxy server to delete.	String
nodeName	Optionally specifies the node in which the proxy server resides.	String

Example usage

```
AdminTask.submitJob('[-jobType deleteProxyServer -jobParams "[[serverName
 testServer1][nodeName testNode1]]" ]')
```

createCluster

The createCluster administrative job creates a cluster. To create the cluster in a cell, you must register the deployment manager of the cell with a job manager.

Job parameters

Parameter	Description	Type
clusterConfig.clusterName	Specifies the name of the server cluster.	String
clusterConfig.preferLocal	Optionally enables node-scoped routing optimization for the cluster.	String
clusterConfig.clusterType	Optionally specifies the type of server cluster.	String
replicationDomain.createDomain	Optionally creates a replication domain with a name set to the name of the new cluster.	String
convertServer.serverNode	Optionally specifies the name of the node of the existing server to convert to the first member of the cluster.	String
convertServer.serverName	Optionally specifies the name of the existing server to convert to the first member of the cluster.	String
convertServer.memberWeight	Optionally specifies the weight value of the new cluster member.	String
convertServer.nodeGroup	Optionally specifies the name of the node group to which all cluster member nodes must belong.	String
convertServer.replicatorEntry	Optionally specifies that a replicator entry for this member is created in the cluster replication domain. The replicator entry is used for HTTP session data replication.	String

Example usage

```
AdminTask.submitJob('[-jobType createCluster -targetList [DmgrManagedNodeName]
 -jobParams "[clusterConfig.clusterName newCluster1]"]')
```

If you specify additional parameters on the createCluster command, use the format of [stepName.parameterName parameterValue] in the -jobParams list of parameters.

```
AdminTask.submitJob('[-jobType createCluster -targetList [DmgrManagedNodeName]
 -jobParams "[ [clusterConfig.clusterName newCluster1] [clusterConfig.clusterType PROXY_SERVER] ]"]')
```

deleteCluster

The deleteCluster administrative job deletes a cluster. To delete the cluster in a cell, you must register the deployment manager of the cell with a job manager.

Job parameters

Parameter	Description	Type
clusterName if you specify a single job parameter, and clusterConfig.clusterName if you specify multiple job parameters.	Specifies the name of the server cluster to delete.	String
replicationDomain.deleteRepDomain	Optionally specifies to delete the cluster replication domain when the cluster is deleted.	String

Example usage

```
AdminTask.submitJob('[-jobType deleteCluster -targetList [DmgrManagedNodeName]
 -jobParams "[clusterName newCluster1]"]')
```

If you specify additional parameters in the deleteCluster command, use the format of [stepName.parameterName parameterValue] in the -jobParams list of parameters.

```
AdminTask.submitJob('[-jobType createCluster -targetList [DmgrManagedNodeName]
 -jobParams "[ [clusterConfig.clusterName newCluster1] [replicationDomain.deleteRepDomain true] ]"]')
```

createClusterMember

The createClusterMember administrative job creates a cluster member on a cluster, which is in a cell. You must register the deployment manager of the cell with a job manager.

Job parameters

Parameter	Description	Type
clusterName	Specifies the name of the server cluster to which the new cluster member belongs.	String
memberConfig.memberNode	Specifies the name of node where the new cluster member is to reside.	String
memberConfig.memberName	Specifies the name of the new cluster member.	String
memberConfig.memberWeight	Optionally specifies the weight of the new cluster member.	String
memberConfig.memberUUID	Optionally specifies the universal unique identifier (UUID) of the new cluster member.	String
memberConfig.genUniquePorts	Optionally specifies to generate unique port numbers for HTTP transports defined in the server.	String
memberConfig.replicatorEntry	Optionally specifies that a replicator entry for this member is created in the cluster replication domain. The replicator entry is used for HTTP session data replication.	String
firstMember.templateName	Optionally specifies the name of the application server template to use as the model for new cluster members.	String
firstMember.templateServerNode	Optionally specifies the name of the node of an existing server to use as a template for new cluster members.	String
firstMember.templateServerName	Optionally specifies the name of the server to use as model for new cluster members.	String
firstMember.nodeGroup	Optionally specifies the name of the node group to which all cluster member nodes must belong.	String
firstMember.coreGroup	Optionally specifies the name of the core group to which all cluster members must belong.	String

Example usage

```
AdminTask.submitJob('[-jobType createClusterMember -targetList
 [DmgrManagedNodeName] -jobParams "[ [memberConfig.memberName newCluster1mem1] [memberConfig.memberNode FederatedNode]
 [clusterName newCluster1] ]"]')
```

deleteClusterMember

The deleteClusterMember administrative job deletes a cluster member from a cluster, which is in a cell. You must register the deployment manager of the cell with a job manager. The cluster must have at least one cluster member on a federated node.

Job parameters

Parameter	Description	Type
clusterName	Specifies the name of a server cluster to which the cluster member to be deleted belongs.	String
memberNode	Specifies the name of the node where the cluster member resides.	String
memberName	Specifies the server name of the cluster member to be deleted.	String
replicatorEntry.deleteEntry	Optionally specifies to delete the replicator entry having the server name of this cluster member from the replication domain of the cluster.	String

Example usage

```
AdminTask.submitJob('[-jobType deleteClusterMember -targetList
 [DmgrManagedNodeName] -jobParams "[ [memberName newCluster1mem1] [memberNode FederatedNode] [clusterName newCluster1]
 ]"]')
```

configureProperties

The configureProperties administrative job applies a properties file to the application server configuration. This job uses the commands in the PropertiesBasedConfiguration command group for the AdminTask object to configure the properties for the target node.

Job parameters

Parameter	Description	Type
propertiesFileLocation	Specifies the location of the properties file to apply to the targeted application server.	String
variableMapLocation	Optionally specifies the location of a variable map to include with the properties file.	String

Example usage

```
AdminTask.submitJob('-jobType configureProperties -targetList
 [Node1,Node2,Node3] -jobParams "[propertiesFileLocation properties\myProperties.props]"')
```

Submit jobs to manage the server runtime

The system administrator can use the application server runtime control jobs to remotely and locally start and stop application servers.

startServer

The startServer administrative job launches and initializes the application server of interest. If a second identical job begins for the same runtime environment and target before the first job finishes, the system merges the subsequent job with the first. The status of the second job reflects the status of the first job that is in progress.

Job parameters

Parameter	Description	Type
serverName	Specifies the application server to start.	String
nodeName	If the target of the job is a deployment manager node, specify the name of the node of interest.	String

Example usage

```
AdminTask.submitJob('-jobType startServer -targetList [Node1,Node2,Node3]
 -jobParams "[serverName AppServer01]"')
```

stopServer

The stopServer administrative job stops the application server of interest. Use the startServer job to restart the application server.

Job parameters

Parameter	Description	Type
serverName	Specifies the application server to stop.	String
nodeName	If the target of the job is a deployment manager node, specify the name of the node of interest.	String

Example usage

```
AdminTask.submitJob('-jobType stopServer -targetList [Node1,Node2,Node3]
 -jobParams "[serverName AppServer01]"')
```

startCluster

The startCluster administrative job starts a cluster. To start a cluster in a cell, you must register the deployment manager of the cell with a job manager.

The startCluster administrative job logs one of the following results when it completes:

- The DISTRIBUTED value specifies that the agent for the node received the job, but the job is not complete.
- The SUCCEEDED value specifies that the job completed successfully.
- The PARTIALLY_SUCCEEDED value specifies that the job was partially completed. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The FAILED value specifies that the job failed.
- The NOT_ATTEMPTED value specifies that the system did not attempt to complete the job.

Job parameters

Parameter	Description	Type
clusterName	Specifies the name of the cluster to start.	String
rippleStart	Optionally specifies whether the cluster is started in a ripple start. The default value is false.	String

Example usage

The following example starts a cluster:

```
AdminTask.submitJob('[-jobType startCluster -targetList [DmgrManagedNodeName]
 -jobParams "[clusterName newCluster1]"]')
```

The following example ripple starts a cluster.

```
AdminTask.submitJob('[-jobType createCluster -targetList [DmgrManagedNodeName]
 -jobParams "[ [clusterName newCluster1] [rippleStart true] ]"]')
```

stopCluster

The stopCluster administrative job stops a cluster. To stop a cluster in a cell, you must register the deployment manager of the cell with a job manager.

The stopCluster administrative job logs one of the following results when it completes:

- The DISTRIBUTED value specifies that the agent for the node received the job, but the job is not complete.
- The SUCCEEDED value specifies that the job completed successfully.
- The PARTIALLY_SUCCEEDED value specifies that the job was partially completed. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The FAILED value specifies that the job failed.
- The NOT_ATTEMPTED value specifies that the system did not attempt to complete the job.

Job parameters

Parameter	Description	Type
clusterName	Specifies the name of the cluster to stop.	String

Example usage

```
AdminTask.submitJob('[-jobType stopCluster -targetList [DmgrManagedNodeName]
 -jobParams "[clusterName newCluster1]"]')
```

AdministrativeJobs command group for the AdminTask object

You can use the Jython scripting language to configure and manage administrative jobs with the wsadmin tool.

Use the following commands to manage administrative jobs for the job manager:

- "deleteJob"
- "getJobTargets" on page 422
- "getJobTargetStatus" on page 422
- "getJobTargetHistory" on page 423
- "getJobTypes" on page 424
- "getJobTypeMetadata" on page 424
- "getOverallJobStatus" on page 425
- "queryJobs" on page 425
- "resumeJob" on page 427
- "submitJob" on page 427
- "suspendJob" on page 428

deleteJob

The deleteJob command deletes an existing job from the job manager. If the job is running when you invoke the command, the system still returns the job results whether or not the job is deleted.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job to delete. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteJob('-jobToken myToken')
```

- Using Jython list:

```
AdminTask.deleteJob('-jobToken', 'myToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteJob('-interactive')
```

getJobTargets

The getJobTargets command displays the target for a job of interest. The target that the command returns for a job might be unenrolled or deleted.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job of interest. (String, required)

Return value

The command returns the node name for the targets for the job of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJobTargets('-jobToken myToken')
```

- Using Jython list:

```
AdminTask.getJobTargets('-jobToken', 'myToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getJobTargets('-interactive')
```

getJobTargetStatus

The getJobTargetStatus command displays the most recent job target status for the job of interest.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job of interest. (String, required)

Optional parameters

-targetList
Specifies a list of target node names. (String [ ], optional)

Return value

The command returns the most recent job status for the targets. The status might be: NOT_ATTEMPTED, DISTRIBUTED, ASYNC_IN_PROGRESS, SUCCEEDED, PARTIALLY_SUCCEEDED, FAILED, DELAYED, or REJECTED.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJobTargetStatus('-jobToken myToken')
```

- Using Jython list:

```
AdminTask.getJobTargetStatus('-jobToken', 'myToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getJobTargetStatus('-interactive')
```

getJobTargetHistory

The getJobTargetHistory command displays the job target history for the job of interest.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job of interest. (String, required)

-target
Specifies the node name of the target of interest. (String, required)

-maxReturn
Specifies the maximum number of results to return. (Integer, required)

Optional parameters

-startingTime
Specifies the time from which the command returns the job target history. (String, optional)

-endingTime
Specifies the time at which the command stops returning the job target history. (String, optional)

-ascending
Specifies whether to return the results in ascending or descending order. Specify `true` to display the results in ascending order, or specify `false` to display the results in descending order. (Boolean, optional)

Return value

The command returns a list of attributes, where the first attribute specifies the number of matches, and the second attribute specifies the history of the job on the target. Each list contains the timestamp, status, message, and result attributes.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJobTargetHistory('-jobToken 2846493472984754 -target 3820J37H3017N294
 -maxReturn 20')
```

- Using Jython list:

```
AdminTask.getJobTargetHistory('-jobToken', '2846493472984754', '-target',
 '3820J37H3017N294', '-maxReturn', '20')
```

Interactive mode example usage

- Using Jython :

```
AdminTask.getJobTargetHistory('-interactive')
```

getJobTypes

The getJobTypes command displays the supported job types for an endpoint of interest.

Target object

None.

Optional parameters

-targetList
Specifies a list of node names for the target. (String [ ], optional)

-group
Specifies the name of the group for the target. (String, optional)

Return value

The command returns a list of job types that each specified target supports.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJobTypes('-target myProfileKey')
```

- Using Jython list:

```
AdminTask.getJobTypes('-target', 'myProfileKey')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getJobTypes('-interactive')
```

getJobTypeMetadata

The getJobTypeMetadata command displays the metadata that is associated with a specific job type.

Target object

None.

Optional parameters

-jobTypeList
Specifies a list of job types of interest. (String [ ], optional)

Return value

The command returns a list of attributes, including the `name`, `label`, `description`, `job-properties`, and `job-parameters` attributes.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJobTypeMetadata('-jobType inventory')
```

- Using Jython list:

```
AdminTask.getJobTypeMetadata('-jobType', 'inventory')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getJobTypeMetadata('-interactive')
```

getOverallJobStatus

The getOverallJobStatus command displays the overall job status for a specific job or a list of jobs of interest.

Target object

None.

Optional parameters

-jobTokenList
Specifies a one or more of the unique identifiers of the jobs of interest. (String [ ], optional)

Return value

The command returns job status information for the job or jobs of interest. The system displays the following information in the overall job status:

- The `STATE` attribute specifies the current state of the job.
- The `TOTAL_RESULTS` attribute specifies the total number of jobs.
- The `DISTRIBUTED` attribute specifies the number of distributed jobs.
- The `ASYNC_IN_PROGRESS` attribute specifies the number of asynchronous jobs in progress.
- The `SUCCEEDED` attribute specifies the number of successful jobs.
- The `PARTIALLY_SUCCEEDED` attribute specifies the number of partially completed jobs. For example, partial success might occur when a node represents multiple servers, and only some of the servers on the node complete the job successfully.
- The `FAILED` attribute specifies the number of failed jobs
- The `NOT_ATTEMPTED` attribute specifies the number of jobs that the system has not attempted.

Batch mode example usage

- Using Jython string:

```
AdminTask.getOverallJobStatus('-jobToken myJobToken')
```

- Using Jython list:

```
AdminTask.getOverallJobStatus('-jobToken', 'myJobToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getOverallJobStatus('-interactive')
```

queryJobs

The queryJobs command queries the job manager for each submitted job.

Target object

None.

Required parameters

-query
Specifies the search expression to use to query for jobs. (String, required)

Use the following guidelines when creating your job queries:

- The query consists of a key, operator, and value, or a list of values. You can specify a single value or a list of values separated by a comma.
- Separate multiple expressions with a space and the `AND` operator.
- The following case sensitive keys are supported:

  jobToken
  Specifies the job token for a specific job to query.

  group Specifies the node group name to query.

  description
  Specifies the description of the job to query. If the description contains multiple words, format the description in single or double quotes such as `description = "job description"`.

  activationDateTime
  Specifies the date and time that the system activates the job, such as `2006-05-03T10:30:45-0000`. The `-0000` section of the activationDateTime key value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `2006-05-03T10:30:45Z`. If you do not specify the time zone, the system uses the time zone of the server.

  expirationDateTime
  Specifies the date and time that the job expires, such as `2006-05-03T10:30:45-0000`. The `-0000` section of the activationDateTime key value represents RFC 822 format. You can specify `Z` as a shortcut for Greenwich Mean Time (GMT), such as `2006-05-03T10:30:45Z`. If you do not specify the time zone, the system uses the time zone of the server.

  state Specifies the state of the job. Valid values include `ASYNC_IN_PROGRESS`, `SUCCEEDED`, `PARTIALLY_SUCCEEDED`, `FAILED`, `DELAYED`, `REJECTED`, and `NOT_ATTEMPTED`.

  target Specifies the target node for a job. Use this key to return the jobs for a specific node. The command returns the jobs for the specific node and node groups that the node belongs to. You can only specify one targetID per query.
- The following operators are supported:

Character	Value
=	Equal to. Specify that the value is null by using `= NULL`.
!=	Not equal to. Specify that the value is not null by using `!= NULL`
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

-maxReturn
Specifies the maximum number of matches to return. (Integer, required)

Return value

The command returns a list of attributes, where the first attribute specifies the number of matches the query produced, and the second attribute specifies a list of job tokens that match the query, as the following sample displays:

```
[ [result [{activationDateTime=2008-03-11T11:56:48-0500,
 expirationDateTime=2008-05-10T11:56:48-0500, jobToken=120525460839085191,
 description=testSubmitJobToValidBaseTargetList}{activationDateTime=2008-03-11T14:05:33-0500,
 expirationDateTime=2008-05-10T14:05:33-0500, jobToken=120526233387582472, description=testSubmitJobToValidBaseTargetList}]] [size
 2] ]
```

Batch mode example usage

- Using Jython string:

```
print AdminTask.queryJobs('-query activationDateTime>= "2006-01-01" activationDateTime<=
 "2007-01-01" -maxReturn 20')

print AdminTask.queryJobs('[-query "target = node3" -maxReturn 2]')
```

- Using Jython list:

```
AdminTask.queryJobs('-query', 'activationDateTime>= "2006-01-01" activationDateTime<=
 "2007-01-01"', '-maxReturn', '20')

print AdminTask.queryJobs(['-query', '"target = node3"', '-maxReturn', '2'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.queryJob('-interactive')
```

resumeJob

The resumeJob command resumes a previously started or suspended job.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job of interest. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.resumeJob('-jobToken myToken')
```

- Using Jython list:

```
AdminTask.resumeJob('-jobToken', 'myToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.resumeJob('-interactive')
```

submitJob

The submitJob command submits a new administrative job to the job manager.

Target object

None.

Required parameters

-jobType
Specifies the type of job to submit. (String, required)

Optional parameters

-group
Specifies the name of the group for the target. (String, optional)

-targetList
Specifies a list of nodes to target. (String [ ], optional)

-jobParams
Specifies the necessary parameters for the job to submit. (Properties, optional)

-username
Specifies the username to use to submit the job when security is enabled. (String, optional)

-password
Specifies the password for the username to use to submit the job when security is enabled. (String, optional)

-description
Specifies a description for the job. (String, optional)

-activationDateTime
Specifies the date and time to activate the job in the format ″2006-05-03T10:30:45-0000″. The ″-0000″ section of the activationDateTime parameter value represents RFC 822 format. You can specify ″Z″ as a shortcut for Greenwich Mean Time (GMT), such as ″2006-05-03T10:30:45Z″. If you do not specify the time zone, the system uses the time zone of the server. (String, optional)

-expirationDateTime
Specifies the expiration date for the job. (String, optional)

-executionWindow
Specifies the recurring interval for the job. (String, optional)

-executionWindowUnit
Specifies the recurring interval unit of measure for the value set by the executionWindow parameter. Specify `DAILY` to run the job daily, `WEEKLY` to run the job weekly, `MONTHLY` to run the job monthly, or `YEARLY` to run the job annually. Additionally, you can specify `CONNECTION` to run the job each time the node connects to the job manager to poll for jobs. When you specify `CONNECTION`, do not set the executionWindow parameter. (String, optional)

-email
Specifies the email address that the system sends job notifications to. (String, optional)

Return value

The command returns a job token for the newly submitted job.

Batch mode example usage

- Using Jython string:

```
AdminTask.submitJob('[-jobType createApplicationServer -target profileKey -jobParams
 "[serverName myServer]" ]')
```

- Using Jython list:

```
AdminTask.submitJob(['-jobType', 'createApplicationServer', '-target', 'profileKey',
 '-jobParams', '"[serverName myServer]"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.submitJob('-interactive')
```

suspendJob

The suspendJob command suspends a job that was previously submitted.

Target object

None.

Required parameters

-jobToken
Specifies the unique identifier of the job to suspend. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.suspendJob('-jobToken myToken')
```

- Using Jython list:

```
AdminTask.suspendJob('-jobToken', 'myToken')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.suspendJob('-interactive')
```

ManagedNodeGroup command group for the AdminTask object

You can use the Jython scripting language to configure managed node groups with the wsadmin tool. Use the commands and parameters in the ManagedNodeGroup command group to create and manage node groups. Create managed node groups to submit jobs from the job manager to one or many managed nodes.

Use the following commands to create and configure managed node groups:

- "addMemberToManagedNodeGroup"
- "createManagedNodeGroup" on page 430
- "deleteManagedNodeGroup" on page 430
- "deleteMemberFromManagedNodeGroup" on page 431
- "getManagedNodeGroupMembers" on page 431
- "getManagedNodeGroupInfo" on page 432
- "queryManagedNodeGroups" on page 432
- "modifyManagedNodeGroupInfo" on page 433

addMemberToManagedNodeGroup

The **addMemberToManagedNodeGroup** command adds a managed node to an existing managed node group.

Target object

None.

Required parameters

-groupName
Specifies the name of the managed node group of interest. (String, required)

-managedNodeNameList
Specifies a list of managed node names to add to the managed node group of interest. (String [ ], required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.addMemberToManagedNodeGroup('-groupName myGroup -managedNodeNameList "[node1, node2, node3]"')
```

- Using Jython list:

```
AdminTask.addMemberToManagedNodeGroup('-groupName', 'myGroup', '-managedNodeNameList', '"[node1, node2, node3]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addMemberToManagedNodeGroup('-interactive')
```

createManagedNodeGroup

The **createManagedNodeGroup** command creates a new managed node group.

Target object

None.

Required parameters

-groupName
Specifies the name of the managed node group of interest. (String, required)

Optional parameters

-description
Specifies a description of the managed node group. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createManagedNodeGroup('-groupName myNewGroup')
```

- Using Jython list:

```
AdminTask.createManagedNodeGroup('-groupName', 'myNewGroup')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createManagedNodeGroup('-interactive')
```

deleteManagedNodeGroup

The **deleteManagedNodeGroup** command deletes a managed node group from your configuration.

Target object

None.

Required parameters

-groupNameList
Specifies a list of managed node groups to delete. (String [ ], required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteManagedNodeGroup('-groupNameList "[group1, group2, group3]"')
```

- Using Jython list:

```
AdminTask.deleteManagedNodeGroup('-groupNameList', '"[group1, group2, group3]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteManagedNodeGroup('-interactive')
```

deleteMemberFromManagedNodeGroup

The **deleteMemberFromManagedNodeGroup** command removes a managed node from a specific managed node group.

Target object

None.

Required parameters

-groupName
Specifies the name of the managed node group of interest. (String, required)

-managedNodeNameList
Specifies a list of managed node Unique Uniform Identifiers (UUID) to delete from the managed node group of interest. (String [ ], required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteMemberFromManagedNodeGroup('-groupName myNewGroup -managedNodeNameList "[node1, node2, node3]"')
```

- Using Jython list:

```
AdminTask.deleteMemberFromManagedNodeGroup('-groupName', 'myNewGroup', '-managedNodeNameList', '"[node1, node2, node3]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteMemberFromManagedNodeGroup('-interactive')
```

getManagedNodeGroupMembers

The **getManagedNodeGroupMembers** command displays the managed nodes that belong to a specific managed node group.

Target object

None.

Required parameters

-groupName
Specifies the name of the managed node group of interest. (String, required)

Return value

The command returns a list of managed node UUIDs that belong to the managed node group of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedNodeGroupMembers('-groupName myNewGroup')
```

- Using Jython list:

```
AdminTask.getManagedNodeGroupMembers('-groupName', 'myNewGroup')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedNodeGroupMembers('-interactive')
```

getManagedNodeGroupInfo

The **getManagedNodeGroupInfo** command displays configuration information for the managed node group of interest.

Target object

None.

Required parameters

-groupName
Specifies the name of one or more managed node groups of interest. (String [ ], required)

Return value

The command returns a list of attributes for each managed node group. Each list of attributes displays the name, size, and description of the managed node group, and whether all group members have an administrative agent.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedNodeGroupInfo('-groupName "[group1, group2, group3]"')
```

- Using Jython list:

```
AdminTask.getManagedNodeGroupInfo('-groupName', '"[group1, group2, group3]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedNodeGroupInfo('-interactive')
```

queryManagedNodeGroups

The **queryManagedNodeGroups** command displays each managed node group in your configuration that meets specific query criteria.

Target object

None.

Required parameters

-maxReturn
Specifies the maximum size of the node group data to display.

Optional parameters

-query
Specifies the settings for which the command queries the managed node groups. You can query for `size`, `description`, `groupName`, and `jobType`. (String, optional)

-validate
Specifies whether to validate the query string. (Boolean, optional)

Return value

The command returns a list of managed node group names.

Batch mode example usage

- Using Jython string:

```
AdminTask.queryManagedNodeGroups('[-maxReturn 10 -query "size=2" -validate true]')
```

- Using Jython list:

```
AdminTask.queryManagedNodeGroups(['-maxReturn', '10', '-query', '"size=2"', '-validate', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.queryManagedNodeGroups('-interactive')
```

modifyManagedNodeGroupInfo

The **modifyManagedNodeGroupInfo** command modifies the description for a managed node group.

Target object

None.

Required parameters

-groupName
Specifies the name of the managed node group of interest. (String, required)

-description
Specifies a new description for the managed node group of interest. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyManagedNodeGroupInfo('-groupName myNewGroup -description "This is my new description of myNewGroup"')
```

- Using Jython list:

```
AdminTask.modifyManagedNodeGroupInfo('-groupName', 'myNewGroup', '-description', '"This is my new description of myNewGroup"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyManagedNodeGroupInfo('-interactive')
```

ManagedNodeAgent command group for the AdminTask object

You can use the Jython scripting language to configure the job manager with the wsadmin tool. Use the commands and parameters in the ManagedNodeAgent group to configure, query, and manage your job manager configuration for managed nodes.

Use the following commands to configure managed node agents for the job manager:

getRuntimeRegistrationProperties

The getRuntimeRegistrationProperties command displays runtime properties for a managed node and respective job manager.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-jobManagerUUID
Specifies the UUID of the job manager of interest. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

Return value

Batch mode example usage

- Using Jython string:

```
AdminTask.getRuntimeRegistrationProperties('-managedNodeName myJobManagedNode
 -jobManagerUUID myJobMgrKey')
```

- Using Jython list:

```
AdminTask.getRuntimeRegistrationProperties('-managedNodeName', 'myJobManagedNode',
 '-jobManagerUUID', 'myJobMgrKey')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getRuntimeRegistrationProperties('-interactive')
```

isPollingJobManager

The isPollingJobManager command determines whether a managed node is polling a job manager.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-jobManagerUUID
Specifies the UUID of the job manager of interest. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.isPollingJobManager('-managedNodeName myJobManagedNode -jobManagerUUID myJobMgrKey')
```

- Using Jython list:

```
AdminTask.isPollingJobManager('-managedNodeName', 'myJobManagedNode', '-jobManagerUUID', 'myJobMgrKey')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isPollingJobManager('-interactive')
```

listJobManagers

The listJobManagers command lists each job manager that a specific managed is registered with.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-jobManagerUUID
Specifies the UUID of the job manager of interest. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

Return value

The command returns a list of job manager properties, including the UUID and host name for the job manager. Depending on the properties defined during managed node registration, the command also might display the port number, connection type, and user name. The value of the password property is not displayed.

Batch mode example usage

- Using Jython string:

```
AdminTask.listJobManagers('-managedNodeName myJobManagedNode')
```

- Using Jython list:

```
AdminTask.listJobManagers('-managedNodeName', 'myJobManagedNode')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listJobManagers('-interactive')
```

registerWithJobManager

The registerWithJobManager command registers a managed node or deployment manager with the job manager.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-host
Specifies the host name of the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the job manager administrative console port number. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

-user
Specifies the user name to log into the job manager. The user must have Administrator role for the job manager. (String, optional)

-password
Specifies the password to log into the job manager. (String, optional)

-alias
Specifies the alias of the managed node to enroll. (String, optional)

-startPolling
Specifies whether the system polls the job manager after it enrolls the managed node. (Boolean, optional)

-autoAcceptSigner
Specifies whether to automatically accept the signer provided by the server. Specify `false` to disable this option. The default value is `true`. (Boolean, optional)

Return value

The command returns the configuration ID of the job manager, as the following output displays:

```
'JobMgr-JOB_MANAGER-2f7d5a29-e601-417b-9124-7737be64dd0a'
```

Batch mode example usage

- Using Jython string:

```
AdminTask.registerWithJobManager('[-host myJobMgrHostname -managedNodeName myJobManagedNode -alias endpoint1]')
```

- Using Jython list:

```
AdminTask.registerWithJobManager(['-host', 'myJobMgrHostname', '-managedNodeName myJobManagedNode', '-alias',
  'endpoint1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.registerWithJobManager('-interactive')
```

setRuntimeRegistrationProperties

The setRuntimeRegistrationProperties command sets runtime properties for managed nodes and job managers.

Target object

None.

Optional parameters

-managedNodeName
Specifies the name of the managed node of interest. If you do not specify the UUID, the system applies the properties to each managed node. (String, optional)

-jobManagerUUID
Specifies the UUID of the job manager of interest. If you do not specify the jobManagerUUID parameter, the system applies the properties to each job manager. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

-interval
Specifies the interval, in seconds, that the system waits before the managed node of interest polls the job manager. (String, optional)

-size
Specifies the maximum size of the thread pool per managed node. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setRuntimeRegistrationProperties('-managedNodeName myJobManagedNode
 -jobManagerUUID myJobMgrKey -interval 600')
```

- Using Jython list:

```
AdminTask.setRuntimeRegistrationProperties('-managedNodeName', 'myJobManagedNode',
 '-jobManagerUUID', 'myJobMgrKey', '-interval', '600')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setRuntimeRegistrationProperties('-interactive')
```

startPollingJobManager

The startPollingJobManager command instructs a managed node to begin polling the job manager.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-jobManagerUUID
Specifies the UUID of the job manager of interest. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.startPollingJobManager('-managedNodeName myJobManagedNode -jobManagerUUID
 myJobMgrKey')
```

- Using Jython list:

```
AdminTask.startPollingJobManager('-managedNodeName', 'myJobManagedNode', '-jobManagerUUID',
 'myJobMgrKey')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.startPollingJobManager('-interactive')
```

stopPollingJobManager

The stopPollingJobManager command instructs a managed node to stop polling the job manager.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-jobManagerUUID
Specifies the UUID of the job manager of interest. (String, optional)

-host
Specifies the host name to use to identify the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the administrative console port number to use to identify the job manager. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.stopPollingJobManager('-managedNodeName myJobManagedNode -jobManagerUUID
 myJobMgrKey')
```

- Using Jython list:

```
AdminTask.stopPollingJobManager('-managedNodeName', 'myJobManagedNode', '-jobManagerUUID',
 'myJobMgrKey')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.stopPollingJobManager('-interactive')
```

unregisterWithJobManager

The unregisterWithJobManager command removes the managed node registration from the job manager configuration.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the managed node of interest. (String, required)

Optional parameters

-host
Specifies the host name of the job manager. The default value is `localhost`. (String, optional)

-port
Specifies the job manager administrative console port number. If security is enabled, use the secure port number. If security is disabled, use the unsecure port number. The default secure port number is `9943`, and the default unsecure port number is `9960`. (String, optional)

-user
Specifies the user name to log into the job manager. The user must have Administrator role for the job manager. (String, optional)

-password
Specifies the password to log into the job manager. (String, optional)

Return value

The command returns the configuration ID of the job manager, as the following output displays:

```
'JobMgr-JOB_MANAGER-0aa85922-bd9a-4ca6-b72c-467cd256b9b3'
```

Batch mode example usage

- Using Jython string:

```
AdminTask.unregisterWithJobManager('[-host myJobMgrHostname -port 8989
 -managedNodeName myJobManagedNode]')
```

- Using Jython list:

```
AdminTask.unregisterWithJobManager(['-host', 'myJobMgrHostname', '-managedNodeName', 'myJobManagedNode'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unregisterWithJobManager('-interactive')
```

JobManagerNode command group for the AdminTask object

You can use the Jython scripting language to manage job manager settings with the wsadmin tool. Use the commands and parameters in the JobManagerNode group to register nodes that do not contain an administrative agent with the job manager.

Use the following commands to administer and query managed node and resource configurations:

cleanupManagedNode

The cleanupManagedNode command cleans up registration information for a managed node. If the system fails when removing a node from the agent, use this command to explicitly clean up the registration information on the job manager. The command does not remove the job history for the node. Jobs in progress continue to run, but new jobs do not start for the node.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the node of interest. (String, required)

Return value

The command returns the UUID of the job manager that the system cleaned up.

Batch mode example usage

- Using Jython string:

```
AdminTask.cleanupManagedNode('-managedNodeName Node1')
```

- Using Jython list:

```
AdminTask.cleanupManagedNode('-managedNodeName', 'Node1')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.cleanupManagedNode('-interactive')
```

getContexts

The getContexts command displays all contexts in the management model, including nodes and servers.

Target object

None.

Required parameters

None.

Return value

The command returns a list of all context paths.

Batch mode example usage

- Using Jython string:

```
AdminTask.getContexts()
```

- Using Jython list:

```
AdminTask.getContexts()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getContexts('-interactive')
```

getManagedNodeKeys

The getManagedNodeKeys command displays the keys to use to query for managed nodes, including the `name`, `alias`, and `uuid` keys.

Target object

None.

Optional parameters

-managedNodeName
Specifies the name of the node of interest. (String, optional)

Return value

The command returns a list of the keys.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedNodeKeys('-managedNodeName Node1')
```

- Using Jython list:

```
AdminTask.getManagedNodeKeys('-managedNodeName', 'Node1')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedNodeKeys('-interactive')
```

getManagedNodeProperties

The getManagedNodeProperties command displays the properties for one or more managed or unmanaged nodes.

Target object

None.

Optional parameters

-managedNodeNameList
Specifies a list of names of the nodes of interest. (String [], optional)

Return value

The command returns a list of properties for each node specified with the -managedNodeNameList parameter.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedNodeProperties('-managedNodeNameList [Node1,Node2]')
```

- Using Jython list:

```
AdminTask.getManagedNodeProperties('-managedNodeNameList', '[Node1, Node2]')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedNodeProperties('-interactive')
```

getManagedResourceProperties

The getManagedResourceProperties command displays the properties of one or more managed resources. Managed resources are instances within a node context or server context. For example, within a server context you can have the managed resources server1, server2, or server3.

Target object

None.

Required parameters

-resourceIdList
Specifies a list of unique identifiers for the resources of interest. (String, optional)

Return value

The command returns a list of properties for each managed resource.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedResourceProperties('-resourceIdList AppSrv01-BASE-b83dc35c-69d4-40af-af60-127de7002cfb/nodes/myNode/servers/server1')
```

- Using Jython list:

```
AdminTask.getManagedResourceProperties('-resourceIdList', 'AppSrv01-BASE-b83dc35c-69d4-40af-af60-127de7002cfb/nodes/myNode/servers/server1')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedResourceProperties('-interactive')
```

getManagedResourcePropertyKeys

The getManagedResourcePropertyKeys command displays the property keys for a specific type of managed resources.

Target object

None.

Required parameters

-resourceType
Specifies the type of managed resource of interest. (String, required)

Return value

The command returns a list of managed resource keys for the specific resource type.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedResourcePropertyKeys('-resourceType server')
```

- Using Jython list:

```
AdminTask.getManagedResourcePropertyKeys('-resourceType', 'server')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedResourcePropertyKeys('-interactive')
```

getManagedResourceTypes

The getManagedResourceTypes command displays each of the managed resource types.

Target object

None.

Required parameters

None.

Return value

The command returns a list of managed resource types.

Batch mode example usage

- Using Jython string:

```
AdminTask.getManagedResourceTypes()
```

- Using Jython list:

```
AdminTask.getManagedResourceTypes()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getManagedResourceTypes('-interactive')
```

modifyManagedNodeProperties

The modifyManagedNodeProperties command replaces each property in a managed node configuration. If the managed node has an administrative agent, the command only modifies the `alias` property. If the managed node does not have an administrative agent, the command replaces all properties.

Target object

None.

Required parameters

-managedNodeName
Specifies the name of the node of interest. (String, required)

-managedNodeProps
Specifies the name and value property pairs to modify for the node of interest. (Properties, required)

Optional parameters

-replace
Specifies whether to replace the existing properties. Specify `true` to replace the existing properties. Specify `false` to merge the properties. The default value is `false`. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyManagedNodeProperties('-managedNodeName Node1 -managedNodeProps "[alias myNewAlias]"')
```

- Using Jython list:

```
AdminTask.modifyManagedNodeProperties('-managedNodeName', 'Node1', '-managedNodeProps', '"[alias myNewAlias]"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyManagedNodeProperties('-interactive')
```

queryManagedNodes

The queryManagedNodes command searches for managed nodes based on a query expression. If you do not specify a query expression, the command returns all managed nodes.

Target object

None.

Required parameters

-maxReturn
Specifies the maximum number of managed nodes to return. (Integer, required)

Optional parameters

-query
Specifies a query that consists of one or more query expressions separated by spaces. If you do not specify this parameter, the command returns all managed nodes. (String, optional)

Construct your queries based on the following guidelines:

- Each query expression consists of the `key`, `operator`, and `value` elements.
- The following operators are supported:

Character	Value
=	Equal to. Specify that the value is null by using `= NULL`.
!=	Not equal to. Specify that the value is not null by using `!= NULL`
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- You can specify one value or multiple values separated by commas.
- You can use pattern matching to specify the value.

-validate
Specifies whether to validate the search query. (Boolean, optional)

Return value

The command returns the number of matches the query found. Secondly, the command returns a list of UUIDs of the managed nodes that met the search query criterion.

Batch mode example usage

- Using Jython string:

```
AdminTask.queryManagedNodes('[-maxReturn 20 -query "alias=managedNode1" -validate true]')
```

- Using Jython list:

```
AdminTask.queryManagedNodes(['-maxReturn', '20', '-query', '"alias=managedNode1"', '-validate', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.queryManagedNodes('-interactive')
```

queryManagedResources

The queryManagedResources command queries your managed resources for specific managed nodes based on a query expression.

Target object

None.

Required parameters

-maxReturn
Specifies the maximum number of managed resources to return. (Integer, required)

Optional parameters

-query
Specifies a query that consists of one or more query expressions separated by spaces. If you do not specify this parameter, the command returns all managed nodes. (String, optional)

Construct your queries based on the following guidelines:

- Each query expression consists of the `key`, `operator`, and `value` elements.
- The following operators are supported:

Character	Value
=	Equal to. Specify that the value is null by using `= NULL`.
!=	Not equal to. Specify that the value is not null by using `!= NULL`
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- You can specify one value or multiple values separated by commas.
- You can use pattern matching to specify the value.

-validate
Specifies whether to validate the search query. (Boolean, optional)

Return value

The command returns the number of matches the query found. Secondly, the command returns a list of UUIDs of the managed nodes that met the search query criterion.

Batch mode example usage

- Using Jython string:

```
AdminTask.queryManagedResources('-maxReturn 20 -query "alias=managedNode1"')
```

- Using Jython list:

```
AdminTask.queryManagedResources('-maxReturn', '20', '-query', '"alias=managedNode1"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.queryManagedResources('-interactive')
```

JobManagerUpkeep command group for the AdminTask object

You can use the Jython scripting language to manage job manager settings with the wsadmin tool. Use the commands and parameters in the JobManagerUpkeep group to back up the job manager database.

Use the following commands to back up your job manager database configuration:

- "backupJobManager"

backupJobManager

The backupJobManager command backs up the job manager database to a specific system location.

Target object

None.

Optional parameters

-location
Specifies the location to which the system saves the backup file. If you do not specify a location, the command saves the job manager backup file to the job manager profile root directory and sets the name of the backup file in the following format: `JobManager_Backup_20080506T053237-828`, where `20080506T053237-828` is a timestamp notation. (String)

-force
Specifies whether to overwrite the backup file if it exists in the location of interest. The default value is `false`. (Boolean)

Return value

The command returns the fully qualified file path of the backup file.

Batch mode example usage

- Using Jython string:

```
AdminTask.backupJobManager('-location /JobManager/backupConfig -force true')
```

- Using Jython list:

```
AdminTask.backupJobManager(['-location', '/JobManager/backupConfig', '-force', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.backupJobManager('-interactive')
```

Managing environment configurations using properties files

Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can extract the configuration attributes and values from your environment to properties files. You can use this functionality for various purposes, including:

- To modify your existing configuration in one location, instead of configuring multiple administrative console panels or running many commands
- To improve the application development life cycle

You can use this topic to manage the following resources in your environment:

- Application servers
- Nodes
- Profiles
- Virtual hosts
- Authorization tables
- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Complete the following steps to extract a properties file for an application server, edit the properties, and apply them to your configuration. You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.commandName('-interactive')
```

Modify an application server configuration, and apply the changes using a properties file.

1. Launch the wsadmin tool.
2. Extract the application server configuration to modify.

   Use the extractConfigProperties command to extract the object configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('-propertiesFileName ConfigProperties_server1.props -configData Server=server1')
```

   The system extracts the properties file, which contains each of the configuration objects and attributes for the `server1` application server.
3. Open the properties file, and manually edit the attribute values of interest.

   Note: Because you are manually editing the properties file, make a back-up copy of the properties file before you edit it.

   The following sample is a section of an application server properties file:

```
#
# Configuration properties file for cells/myCell/nodes/myNode/servers/server1|server.xml#
# Extracted on Thu Sep 06 00:27:26 CDT 2007
#
#
# Section 1.0 ## cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
# SubSection 1.0 # Server Section
#
ResourceType=Server
ImplementingResourceType=Server
ResourceId=cells/myCell/nodes/myNode/servers/server1|server.xml#server1
```

```
#
#
#Properties
#
shortName=null
serverType=APPLICATION_SERVER
developmentMode=false #boolean
name=server1
parallelStartEnabled=true #boolean
clusterName=C
modelId=null
uniqueId=null
#
```

To modify the application server to run in development mode and disable parallel start, modify the `developmentMode` and `parallelStartEnabled` properties, as the following example demonstrates:

```
#
# Configuration properties file for cells/myCell/nodes/myNode/servers/server1|server.xml#
# Extracted on Thu Sep 06 00:27:26 CDT 2007
#
#
# Section 1.0 ## cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
# SubSection 1.0 # Server Section
#
ResourceType=Server
ImplementingResourceType=Server
ResourceId=cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
#Properties
#
shortName=null
serverType=APPLICATION_SERVER
developmentMode=true #boolean
name=server1
parallelStartEnabled=false #boolean
clusterName=C
modelId=null
uniqueId=null
#
```

4. Validate the properties file.

   Note: As a best practice, use the validateConfigProperties command to validate the modified properties file before applying the changes, as the following Jython example demonstrates:

   ```
   AdminTask.validateConfigProperties('-propertiesFileName ConfigProperties_server1.props -reportFile report.txt')
   ```

   The command returns a value of `true` if the system successfully validates the properties file. The command returns a value of `false` if the system does not validate the file.

5. Apply the changes to the application server.

   Use the applyConfigProperties command to apply the changes to the application server.

   ```
   AdminTask.applyConfigProperties('-propertiesFileName ConfigProperties_server1.props -validate true')
   ```

6. Save your configuration changes.

   ```
   AdminConfig.save()
   ```

Related tasks

“Extracting properties files”
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Applying properties files” on page 455
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Validating properties files” on page 453
Use this topic to validate configuration properties before applying properties files to your configuration.

“Creating server, cluster, application, or authorization group objects using properties files” on page 458
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Deleting server, cluster, application, or authorization group objects using properties files” on page 460
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Creating and deleting configuration objects using properties files” on page 462
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Extracting properties files

Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can extract the configuration attributes and values from your environment to properties files.

This topic You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.commandName('-interactive')
```

- Extract a cell configuration.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Extract the cell configuration.

     Use the extractConfigProperties command to extract the object configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName
 ConfigProperties_cell.props]')
```

     The system extracts the properties file, as the following example displays:

```
Cell.props  # # SubSection 1.0 # Cell level attributes # ResourceType=Cell
 ImplementingResourceType=Cell ResourceId=Cell=!{cellName} #  # #Properties # shortName=null cellType=DISTRIBUTED
 #ENUM(UDP|TCP|MULTICAST|DISTRIBUTED|STANDALONE),readonly name=!{cellName} multicastDiscoveryAddressEndpointName=null
 discoveryAddressEndpointName=null cellDiscoveryProtocol=TCP #ENUM(UDP|TCP|MULTICAST)  .... ...  Properties of nodes,servers,
 clusters, applications, etc. ....    EnvironmentVariablesSection # # #Environment Variables #Day Month 17 Time CDT Year
 cellName=myCell
```

The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

- Extract a server configuration.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Extract the application server configuration of interest.

     Use the extractConfigProperties command to extract the server configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName ConfigProperties_server1.props -configData Server=server1]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Server Section # ResourceType=Server ImplementingResourceType=Server
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName} #  # #Properties # shortName=null serverType=DEPLOYMENT_MANAGER
 #readonly developmentMode=false #boolean parallelStartEnabled=true #boolean name=!{serverName} clusterName=null uniqueId=null
 modelId=null  ... ... Properties of other inner objects ( EJBContainer, WebContainer, ORB etc) and subtypes not shown. ...
 EnvironmentVariablesSection # #Environment Variables #Day Month 16 Time CDT Year cellName=myCell nodeName=myNode
 hostName=myHost.com serverName=dmgr
```

     The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

- Extract the a server subtype configuration for a specific server.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Extract the EJB container and Web container properties for a specific server.

     Use the extractConfigProperties command to extract the server configuration, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName ejbcontainer.props -configData
 Server=server1 -filterMechanism SELECTED_SUBTYPES -selectedSubTypes [EJBContainer WebContainer]]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # EJBContainer # ResourceType=EJBContainer
 ImplementingResourceType=EJBContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBCon
ntainer_1 AttributeInfo=components #  # #Properties # EJBTimer={} #ObjectName*(null) name=null defaultDatasourceJNDIName=null
 inactivePoolCleanupInterval=30000 #long passivationDirectory="${USER_INSTALL_ROOT}/temp" enableSFSBFailover=false #boolean
 server=null parentComponent=Network Deployment Server  # # SubSection 1.0 # WebContainer  # ResourceType=WebContainer
 ImplementingResourceType=WebContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:WebContainer=ID#WebCon
ntainer_1 AttributeInfo=components #  # #Properties # enableServletCaching=false #boolean name=null defaultVirtualHostName=null
 server=null maximumPercentageExpiredEntries=15 #integer asyncIncludeTimeout=60000 #integer parentComponent=Network Deployment
 Server disablePooling=false #boolean sessionAffinityFailoverServer=null maximumResponseStoreSize=100 #integer
 allowAsyncRequestDispatching=false #boolean sessionAffinityTimeout=0 #integer   EnvironmentVariablesSection # #Environment
 Variables #Thu Apr 17 14:17:25 CDT 2008 cellName=myCell nodeName=myNode hostName=myhost.com serverName=dmgr
```

     The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

     The EJBContainer=ID#EJBContainer_1 string represents the EJBContainer object within the server. Use this XML ID to uniquely identify the object in the configuration. You can modify this field to EJBContainer=myContainer if the name field is set to myContainer in the configuration before you apply the properties file to the configuration.

- Extract node properties without traversing the subtypes of the node.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Extract the node properties, except for specific subtype properties of servers and resources.

     Use the extractConfigProperties command to extract the node configuration properties, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName node.props -configData
 Node=myNode -filterMechanism NO_SUBTYPES]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Node Section # ResourceType=Node ImplementingResourceType=Node
 ResourceId=Cell=!{cellName}:Node=!{nodeName} #  # #Properties # shortName=null name=!{nodeName}
 maxFilePermissionForApps=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755 " discoveryProtocol=TCP #ENUM(UDP|TCP|MULTICAST)
 hostName=!{hostName}  # ## Section 1.0_1#Cell=!{cellName}:Node=!{nodeName} # ResourceType=Node ImplementingResourceType=Node
 ExtensionId=NodeMetadataExtension ResourceId=Cell=!{cellName}:Node=!{nodeName}  # nodeOS=distributed nodeVersion=7.0.0.0   # #
 End of Section 1.0_1# Cell=!{cellName}:Node=!{nodeName} #  # # End of Section 1.0# Cell=!{cellName}:Node=!{nodeName} #
 EnvironmentVariablesSection # #Environment Variables #Day Month 17 Time CDT Year cellName=myCell nodeName=myNode
```

The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as `!{cellName}`, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section of the properties file contains each variable in the file.

- Extract node properties without traversing the subtypes of the node or invoking extensions.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Extract the node properties, except for specific subtype properties of servers and resources and without invoking extensions.

     Use the extractConfigProperties command to extract the node configuration properties, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName node.props -configData
 Node=myNode -filterMechanism NO_SUBTYPES_AND_EXTENSIONS]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Node Section # ResourceType=Node ImplementingResourceType=Node
 ResourceId=Cell=!{cellName}:Node=!{nodeName} #  # #Properties # shortName=null name=!{nodeName}
 maxFilePermissionForApps=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755 " discoveryProtocol=TCP #ENUM(UDP|TCP|MULTICAST)
 hostName=!{hostName}  # ## Section 1.0_1#Cell=!{cellName}:Node=!{nodeName} # ResourceType=Node ImplementingResourceType=Node
 ExtensionId=NodeMetadataExtension ResourceId=Cell=!{cellName}:Node=!{nodeName}  # nodeOS=distributed nodeVersion=7.0.0.0   # #
 End of Section 1.0_1# Cell=!{cellName}:Node=!{nodeName} #  # # End of Section 1.0# Cell=!{cellName}:Node=!{nodeName} #
 EnvironmentVariablesSection # #Environment Variables #Day Month 17 Time CDT Year cellName=myCell nodeName=myNode
```

     The command excludes the NodeMetadataExtension section from the extracted properties file, as that is an extension to a node resource. The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as `!{cellName}`, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section of the properties file contains each variable in the file.

What to do next

After extracting properties files, use this functionality for various purposes, including:

- To modify your existing configuration in one location, instead of configuring multiple administrative console panels or running many commands
- To improve the application development life cycle

You can use properties files to manage the following server subtypes in your environment:

- Application servers
- Nodes
- Profiles
- Virtual hosts
- Authorization tables
- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Related tasks

“Validating properties files”
Use this topic to validate configuration properties before applying properties files to your configuration.

“Applying properties files” on page 455
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Creating server, cluster, application, or authorization group objects using properties files” on page 458
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Deleting server, cluster, application, or authorization group objects using properties files” on page 460
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Creating and deleting configuration objects using properties files” on page 462
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Validating properties files

Use this topic to validate configuration properties before applying properties files to your configuration.

Before you begin

Use the extractConfigProperties command in the PropertiesBasedConfiguration command group to extract a properties file from your configuration. Use a text editor to modify the properties in the properties file.

About this task

Note: There are two steps to validate a properties file before applying it to the configuration. First, use the validateConfigProperties command to validate the properties file. Then, use the applyConfigProperties command and the -validate option to apply the properties and validate the file simultaneously.

Use the validateConfigProperties command to validate a properties file.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Validate the properties file of interest.

   For this example, validate the following EJBContainer properties file:

```
# # SubSection 1.0 # EJBContainer # ResourceType=EJBContainer
 ImplementingResourceType=EJBContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}
 :ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBCon
ntainer_1 AttributeInfo=components # # #Properties # EJBTimer={}
```

```
#ObjectName*(null) name=null defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long passivationDirectory="${USER_INSTALL_ROOT}/temp"
 enableSFSBFailover=true#boolean server=null parentComponent=Network Deployment Server
```

Always validate the entire properties file. Do not validate subsections of files. Use the validateConfigProperties command to validate the properties file, as the following Jython example demonstrates:

```
AdminTask.validateConfigProperties('[-propertiesFileName ejbcontainer.props
 -variablesMapFileName ejbprops.vars -reportFileName report.txt]')
```

The command returns a value of `true` if the system successfully validates the properties file. The command returns a value of `false` if the system does not validate the file.

The command also generates a report file and records configuration actions such as:

- changes to property values.
- no change to property values when the configuration value is the same as defined in the properties file.
- no change to read-only property values.
- exceptions.

The following example displays a sample report file:

```
ADMG0820I: Start applying properties from file ejbcontainer.props ADMG0818I: Processing
 section EJBContainer:ApplicationServer. ADMG0810I: Not changing value for this property EJBTimer. New value specified is same as
 current value {}. ADMG0810I: Not changing value for this property defaultDatasourceJNDIName. New value specified is same as
 current value null. ADMG0811I: Changing value for this property enableSFSBFailover. New value specified is true. Old value was
 false. ADMG0810I: Not changing value for this property inactivePoolCleanupInterval. New value specified is same as current value
 30000. ADMG0810I: Not changing value for this property name. New value specified is same as current value null. ADMG0807I:
 Property parentComponent is readonly. Will not be modified ADMG0810I: Not changing value for this property passivationDirectory.
 New value specified is same as current value ${USER_INSTALL_ROOT}/temp. ADMG0807I: Property server is readonly. Will not be
 modified ADMG0819I: End Processing section EJBContainer:ApplicationServer.
```

To make the reports more concise, specify the reportFilterMechanism parameter with the validateConfigProperties command to only report errors and changes to the configuration, as the following example demonstrates:

```
AdminTask.validateConfigProperties('[-propertiesFileName ejbcontainer.props
 -variablesMapFileName ejbprops.vars -reportFileName report.txt -reportFilterMechanism
 Errors_And_Changes]')
```

The filtered report file displays error and configuration changes only, as the following sample output demonstrates:

```
ADMG0820I: Start applying properties from file ejbcontainer.props ADMG0811I: Changing value
 for this property enableSFSBFailover. New value specified is true. Old value was false.  AADMG0831E: Value specified for property
 inactivePoolCleanupInterval is not a valid type. Specified value asdf, Required type long. ADMG0821I: End applying properties
 from file ejbcontainer.props.
```

Related tasks

“Applying properties files”
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Creating server, cluster, application, or authorization group objects using properties files” on page 458
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Creating and deleting configuration objects using properties files” on page 462
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Deleting server, cluster, application, or authorization group objects using properties files” on page 460
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files” on page 450
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Applying properties files

Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

Before you begin

Use the extractConfigProperties command in the PropertiesBasedConfiguration command group to extract the properties files of interest. Use a text editor to modify one or more values in the properties file.

Use the validateConfigProperties command in the PropertiesBasedConfiguration command group to validate the modified properties file before applying the file to your configuration.

About this task

You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.commandName('-interactive')
```

- Modify one or more properties and apply the properties file to the configuration.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Modify the properties of interest.

     In the following properties file, use a text editor to change the value of the `enableSFSB` property:

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBContainer_1
AttributeInfo=components
#

#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=true #boolean
server=null
parentComponent=Network Deployment Server

EnvironmentVariablesSection
#
#
#Environment Variables
#Thu Apr 17 14:10:31 CDT 2008
hostName2=*
hostName1=localhost
cellName=IBM-49F7FB781FECell07
nodeName=IBM-49F7FB781FECellManager07
hostName=IBM-49F7FB781FE.austin.ibm.com
serverName=dmgr
enableSSB=true
```

3. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props]')
```

- Use additional user modified variables to modify the configuration.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Use additional variables to modify the enableSFSBFailover property of the EJB container, changing the value from `true` to `false`.

     In the following properties file, modify the enableSFSBFailover property by specifying the value as the `!{enableSSB}` variable. You can use the variable in the section header or in the properties part of the section. Also, one property value can contain multiple variables as shown for ResourceId.

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBContainer_1
AttributeInfo=components
#

#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=!{enableSSB} #boolean
server=null
parentComponent=Network Deployment Server

EnvironmentVariablesSection
#
#
#Environment Variables
#Thu Apr 17 14:10:31 CDT 2008
hostName2=*
hostName1=localhost
```

```
cellName=IBM-49F7FB781FECell07
nodeName=IBM-49F7FB781FECellManager07
hostName=IBM-49F7FB781FE.austin.ibm.com
serverName=dmgr
enableSSB=true
```

3. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props]')
```

- Modify the configuration by applying a properties file and a variable map.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Modify the enableSFSBFailover property of the EJB container, changing the value from `true` to `false`.

     Modify the enableSFSBFailover property by specifying the value as the `!{enableSSB}` variable in a separate variable map file. Instead of specifying the variable in the section header or in the properties part of the section, create a separate variable map file. The following code displays a sample variable map file:

```
ejbprops.vars:
#
#
#Environment Variables
#Day Month 11 Time CDT Year
hostName2=*
hostName1=localhost
cellName=myCell
nodeName=myNode
hostName=myhost.com
serverName=myServer
enableSSB=true
```

     The following code displays the corresponding properties file to apply to the configuration:

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBContainer_1
AttributeInfo=components
#
#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=true#boolean
server=null
parentComponent=Network Deployment Server
```

  3. Apply the modified properties to your configuration.

     Use the applyConfigProperties command to apply the properties file and the variable map file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props -variablesMapFileName ejbprops.vars]')
```

What to do next

To verify that the system made the changes to your configuration, extract the properties file from your configuration using the extractPropertiesFile command.

Related tasks

“Extracting properties files” on page 450
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Validating properties files” on page 453
Use this topic to validate configuration properties before applying properties files to your configuration.

“Creating server, cluster, application, or authorization group objects using properties files”
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Deleting server, cluster, application, or authorization group objects using properties files” on page 460
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Creating and deleting configuration objects using properties files” on page 462
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Creating server, cluster, application, or authorization group objects using properties files

Use this topic to create new server, cluster, application, or authorization group objects your configuration.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to create configuration objects in your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Create a properties file template.

   Create a properties file template to use to create the new server, cluster, application, or authorization group object. Use the -configType parameter and the following guidelines to specify the type of template to create:

   - Specify `Server` to create a server type properties file template.
   - Specify `ServerCluster` to create a server cluster type properties file template.
   - Specify `Application` to create an application type properties file template.
   - Specify `AuthorizationGroup` to create an authorization group type properties file template.

   The following Jython example uses the createPropertiesFileTemplates command to create a new AuthorizationGroup object template:

```
AdminTask.createPropertiesFileTemplates('[-propertiesFileName authorizationGroup.template -configType AuthorizationGroup]')
```

The command generates a template file similar to the following sample template:

```
#
# Create parameters
# Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke applyConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=
SKIP=true
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=authorizationGroupName #String,required
commandName=createAuthorizationGroup
```

3. Modify the new template file.

   Modify the new AuthorizationGroup template file by setting the required parameters. You can also modify the optional parameters, but you must modify the required parameters. Change the `SKIP` required property value from `SKIP=true` to `SKIP=false` to indicate that the system should apply the properties in the specific section of the properties file to the configuration. To ignore a specific section of a properties file, set the `SKIP` property to `SKIP=true`.

```
#
# Create parameters
# Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke applyConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=
SKIP=false
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=ag1 #String,required
commandName=createAuthorizationGroup
```

4. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName authorizationGroup.template]')
```

   The command creates the `ag1` authorization group in your configuration.

5. Save the configuration changes.

Related tasks

“Extracting properties files” on page 450
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Applying properties files” on page 455
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Validating properties files” on page 453
Use this topic to validate configuration properties before applying properties files to your configuration.

“Deleting server, cluster, application, or authorization group objects using properties files”
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Creating and deleting configuration objects using properties files” on page 462
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Deleting server, cluster, application, or authorization group objects using properties files

Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to delete configuration objects from your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Create a properties file template.

   Create a properties file template to use to delete the server, cluster, application, or authorization group object of interest. Use the -configType parameter and the following guidelines to specify the type of template to create:

   - Specify `Server` to create a server type properties file template.
   - Specify `ServerCluster` to create a server cluster type properties file template.
   - Specify `Application` to create an application type properties file template.
   - Specify `AuthorizationGroup` to create an authorization group type properties file template.

   The following Jython example uses the createPropertiesFileTemplates command to create a new AuthorizationGroup object template:

```
AdminTask.createPropertiesFileTemplates('[-propertiesFileName authorizationGroup.template -configType AuthorizationGroup]')
```

The command generates a template file similar to the following sample template:

```
#
# Delete parameters
# Replace the line 'SKIP=true' with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke deleteConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=
SKIP=true
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=authorizationGroupName #String,required
commandName=deleteAuthorizationGroup
```

3. Modify the new template file.

   Modify the new AuthorizationGroup template file by setting the required parameters. You can also modify the optional parameters, but you must modify the required parameters.

   Change the `SKIP` required property value from `SKIP=true` to `SKIP=false` to indicate that the system should apply the properties in the specific section of the properties file to the configuration. To ignore a specific section of a properties file, set the `SKIP` property to `SKIP=true`.

```
#
# Delete parameters
# Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke deleteConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=authorizationGroupName
SKIP=false
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=authorizationGroupName #String,required
commandName=deleteAuthorizationGroup
```

4. Remove the object from your configuration.

   Use the deleteConfigProperties command to remove the existing AuthorizationGroup object from the configuration, as the following Jython example demonstrates:

```
AdminTask.deleteConfigProperties('[-propertiesFileName authorizationGroup.template]')
```

   The command removes the `ag1` authorization group in your configuration.

5. Save the configuration changes.

Related tasks

“Creating server, cluster, application, or authorization group objects using properties files” on page 458
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Creating and deleting configuration objects using properties files”
Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

“Validating properties files” on page 453
Use this topic to validate configuration properties before applying properties files to your configuration.

“Applying properties files” on page 455
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files” on page 450
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Creating and deleting configuration objects using properties files

Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to create and delete configuration objects from your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Extract a properties file for the subtype of interest from your configuration.

   Use the extractConfigProperties command to extract the properties file for the resource of interest. The following example extracts the properties for the ThreadPool resource:

```
AdminTask.extractConfigProperties('[-propertiesFileName threadPool.props -configData
 Server=server1 -filterMechanism SELECTED_SUBTYPES -selectedSubTypes [ThreadPool]]')
```

   The command generates a template file similar to the following sample template:

```
# # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
 ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
 eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=ID#builtin_ThreadPool_4 #  # #Properties # maximumSize=20 #integer name=Default
 inactivityTimeout=5000 #integer minimumSize=5 #integer isGrowable=false #boolean
```

3. Create or delete configuration objects.

   To create a new thread pool or delete the existing thread pool, modify the ResourceId attribute.

- To create a new thread pool, set the ResourceId attribute to a value that does not exist in your configuration. In the following example, note that the ThreadPool=ID#builtin_ThreadPool_4 ResourceId is replaced with the ThreadPool=ID#ThreadPool_99999 ResourceId, which does not exist in the configuration:

```
# # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=ID#ThreadPool_99999 #  # #Properties # maximumSize=20
#integer name=myThreadPool inactivityTimeout=5000 #integer minimumSize=5 #integer isGrowable=false #Boolean
```

  Run the applyConfigProperties command to apply the properties file to your configuration, as the following command demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName threadPool.props]')
```

  The command automatically validates the properties file, then uses the modified values in the file to create a new thread pool in your configuration.

- To delete the thread pool, specify the DELETE=true property in the header of the properties file, as the following example demonstrates:

```
# # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
 ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
 eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=myThreadPool DELETE=true #  # #Properties # maximumSize=20 #integer
 name=myThreadPool inactivityTimeout=5000 #integer minimumSize=5 #integer isGrowable=false #boolean
```

  Run the deleteConfigProperties command to use the properties file to remove the thread pool from your configuration, as the following command demonstrates:

```
AdminTask.deleteConfigProperties('[-propertiesFileName threadPool.props]')
```

  The command automatically validates the properties file, then uses the new attribute and value in the file to remove the thread pool from your configuration.

  Note: If you run the deleteConfigProperties command before you add the DELETE=true attribute and value to the properties file, the command resets each property to the default value. The system completely removes properties that do not have default values.

4. Save the configuration changes.

Related tasks

“Extracting properties files” on page 450
Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

“Applying properties files” on page 455
Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

“Validating properties files” on page 453
Use this topic to validate configuration properties before applying properties files to your configuration.

“Creating server, cluster, application, or authorization group objects using properties files” on page 458
Use this topic to create new server, cluster, application, or authorization group objects your configuration.

“Deleting server, cluster, application, or authorization group objects using properties files” on page 460
Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

“Managing environment configurations using properties files” on page 447
Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

“Extracting properties files to troubleshoot your environment” on page 1174
Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

Chapter 8, “Managing servers and nodes with scripting,” on page 401
Use the wsadmin tool to administer your administrative architecture and runtime settings.

Related reference

“PropertiesBasedConfiguration command group for the AdminTask object” on page 475
You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Stopping a node using scripting

Use scripting to stop a node agent.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Stopping the node agent on a remote machine process is an asynchronous action where the stop is initiated, and then control returns to the command line. Perform the following task to stop a node:

1. Identify the node that you want to stop and assign it to a variable:

   Using Jacl:

   ```
   set na [$AdminControl queryNames type=NodeAgent,node=mynode,*]
   ```

   Using Jython:

   ```
   na = AdminControl.queryNames('type=NodeAgent,node=mynode,*')
   ```

2. Stop the node:

   Using Jacl:

   ```
   $AdminControl invoke $na stopNode
   ```

   Using Jython:

```
AdminControl.invoke(na, 'stopNode')
```

Restarting node agent processes using the wsadmin tool

If you stop a node agent process, you cannot start the process using the wsadmin tool or the administrative console. Use this topic to restart a node agent that has been stopped.

Before you begin

A node agent must exist in your configuration and must be in the stopped state.

About this task

Use the following steps to restart a node agent process:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the name of the node agent to restart.
   - Using Jacl:
     ```
     set na [$AdminControl queryNames type=NodeAgent,node=mynode,*]
     ```
   - Using Jython:
     ```
     na = AdminControl.queryNames('type=NodeAgent,node=mynode,*')
     ```
3. Determine the NodeAgent MBean operation.

   Use the following help commands to return information about the restart option:
   - Using Jacl:
     ```
     $Help operations $na
     ```
   - Using Jython:
     ```
     print Help.operations('na')
     ```
4. Restart the node agent process.

   As the help output from the operations command displays, you must specify two boolean parameters in the command invocation. First, specify `true` for the syncFirst parameter to synchronize your configuration before the command restarts the node. Next, specify `true` for the restartServers parameter to restart all running servers while the command restarts the node. The following command example synchronizes and restarts the application servers that are running on the node when the node agent restarts:
   - Using Jacl:
     ```
     $AdminControl invoke $na restart "true true"
     ```
   - Using Jython:
     ```
     AdminControl.invoke(na,'restart','true true')
     ```

Results

The node agent process has been restarted.

What to do next

Verify that your node agent and servers successfully started.

Starting servers using scripting

You can use scripting and the wsadmin tool to start servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

Use the **startServer** command to start the server. This command has several syntax options. For example:

- To start a server on a WebSphere Application Server single server edition, choose one of the following options:
  - The following examples specify the server name only:

    Using Jacl:

    ```
    $AdminControl startServer serverName
    ```

    Using Jython:

    ```
    AdminControl.startServer('serverName')
    ```
  - The following example starts an application server with the node specified:
    - Using Jacl:

      ```
      $AdminControl startServer server1 mynode
      ```
    - Using Jython:

      ```
      print AdminControl.startServer('server1', 'mynode')
      ```

    Example output:

    ```
    WASX7319I: The serverStartupSyncEnabled attribute is set to false.  A start
    will be attempted for server "server1" but the configuration information for
    node "mynode" may not be current.
    WASX7262I: Start completed for server "server1" on node "mynode"
    ```
  - The following example specify the server name and wait time:
    - Using Jacl:

      ```
      $AdminControl startServer serverName 10
      ```
    - Using Jython:

      ```
      AdminControl.startServer('serverName', 10)
      ```

    where *10* is the number of seconds that the process should wait before starting the server.
- To start a server on a WebSphere Application Server network deployment edition, choose one of the following options:
  - The following example specifies the server name and the node name:
    - Using Jacl:

      ```
      $AdminControl startServer serverName nodeName
      ```
    - Using Jython:

      ```
      AdminControl.startServer('serverName', 'nodeName')
      ```
  - The following example specifies the server name, the node name, and the wait time:
    - Using Jacl:

      ```
      $AdminControl startServer serverName nodeName 10
      ```
    - Using Jython:

      ```
      AdminControl.startServer('serverName', 'nodeName', 10)
      ```

    where *10* is the number of seconds that the process should wait before starting the server.

Stopping servers using scripting

You can stop servers using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

Use the **stopServer** command to stop the server. This command has several syntax options. For example:

- To stop a server on a WebSphere Application Server single server edition, choose one of the following options:
  - The following examples specify the server name only:

    Using Jacl:

    ```
    $AdminControl stopServer serverName
    ```

    Using Jython:

    ```
    AdminControl.stopServer('serverName')
    ```
  - The following examples stop an application server with the node specified:
    - Using Jacl:

      ```
      $AdminControl stopServer serverName mynode
      ```
    - Using Jython:

      ```
      print AdminControl.stopServer('serverName', 'mynode')
      ```

    Example output:

    ```
    WASX7337I: Invoked stop for server "serverName" Waiting for stop completion.
    WASX7264I: Stop completed for server "serverName" on node "mynode"
    ```
  - The following examples specify the server name and immediate:
    - Using Jacl:

      ```
      $AdminControl stopServer serverName immediate
      ```
    - Using Jython:

      ```
      AdminControl.stopServer('serverName', immediate)
      ```
- To stop a server on a WebSphere Application Server network deployment edition, choose one of the following options:
  - The following example specifies the server name and the node name:
    - Using Jacl:

      ```
      $AdminControl stopServer serverName nodeName
      ```
    - Using Jython:

      ```
      AdminControl.stopServer('serverName', 'nodeName')
      ```
  - The following example specifies the server name, the node name, and immediate:
    - Using Jacl:

      ```
      $AdminControl stopServer serverName nodeName immediate
      ```
    - Using Jython:

      ```
      AdminControl.stopServer('serverName', 'nodeName', immediate)
      ```

Querying server state using scripting

You can use the wsadmin tool and scripting to query server states.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

When querying the server state, the following command steps return a value of STARTED if the server is started. If the server is stopped, the command does not return a value.

In a Network Deployment environment, you also can query for the server status from the deployment manager. If the server is active, the command returns the STARTED return value. If the server is stopped, the command returns the STOPPED return value.Perform the following steps to query the server state:

- Identify the server and assign it to the server variable. The following example returns the server MBean that matches the partial object name string:
  - Using Jacl:

    ```
    set server [$AdminControl completeObjectName cell=mycell,node=mynode,
    name=server1,type=Server,*]
    ```

  - Using Jython:

    ```
    server = AdminControl.completeObjectName('cell=mycell,node=mynode,
    name=server1,type=Server,*')
    print server
    ```

    Example output:

    ```
    WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#Server_1,
    type=Server,node=mynode,process=server1,processType=ManagedProcess
    ```

    If the server is stopped, the **completeObjectName** command returns an empty string (' ').
- Query for the state attribute. In addition to using the previous step, you can also query for the server state attribute. For example:
  - Using Jacl:

    ```
    $AdminControl getAttribute $server state
    ```

  - Using Jython:

    ```
    print AdminControl.getAttribute(server, 'state')
    ```

    The **getAttribute** command returns the value of a single attribute.

    Example output:

    ```
    STARTED
    ```

Listing running applications on running servers using scripting

Use the wsadmin tool and scripting to list all the running applications on all the running servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Use the following example to list all the running applications on all the running servers on each node of each cell:

- Using Jacl:

```
*Provide this example as a Jacl script file and run it with  the "-f" option:
1  #-------------------------------------------------------------------
2  # lines 4 and 5 find all the cell and process them one at a time
3  #-------------------------------------------------------------------
4  set cells [$AdminConfig list Cell]
5  foreach cell $cells {
6     #-----------------------------------------------------------------------
7     # lines 10 and 11 find all the nodes belonging to the cell and
8     # process them at a time
```

```
9     #-------------------------------------------------------------------
10    set nodes [$AdminConfig list Node $cell]
11    foreach node $nodes {
12       #-------------------------------------------------------------
13       # lines 16-20 find all the running servers belonging to the cell
14       # and node, and process them one at a time
15       #-------------------------------------------------------------
16       set cname [$AdminConfig showAttribute $cell name]
17       set nname [$AdminConfig showAttribute $node name]
18       set servs [$AdminControl queryNames type=Server,cell=$cname,node=$nname,*]
19       puts "Number of running servers on node $nname: [llength $servs]"
20       foreach server $servs {
21           #-----------------------------------------------------
22           # lines 25-31 get some attributes from the server to display;
23           # invoke an operation on the server JVM to display a property.
24           #-----------------------------------------------------
25           set sname [$AdminControl getAttribute $server name]
26           set ptype [$AdminControl getAttribute $server processType]
27           set pid   [$AdminControl getAttribute $server pid]
28           set state [$AdminControl getAttribute $server state]
29           set jvm [$AdminControl queryNames type=JVM,cell=$cname,
       node=$nname,process=$sname,*]
30           set osname [$AdminControl invoke $jvm getProperty os.name]
31           puts "  $sname ($ptype) has pid $pid; state: $state; on $osname"
32
j3           #-----------------------------------------------------
34           # line 37-42 find the applications running on this server and
35           # display the application name.
35           #-----------------------------------------------------
37           set apps [$AdminControl queryNames type=Application,
       cell=$cname,node=$nname,process=$sname,*]
38           puts "  Number of applications running on $sname: [llength $apps]"
39           foreach app $apps {
40              set aname [$AdminControl getAttribute $app name]
41              puts "    $aname"
42           }
43           puts "-----------------------------------------------------"
44           puts ""
45
46         }
47      }
48  }
```

- Using Jython:

  * Provide this example as a Jython script file and run it with the "-f" option:

```
1   #-------------------------------------------------------------------
2   # lines 7 and 8 find all the cell and process them one at a time
3   #-------------------------------------------------------------------
4   # get line separator
5   import  java.lang.System  as  sys
6   lineSeparator = sys.getProperty('line.separator')
7   cells = AdminConfig.list('Cell').split(lineSeparator)
8   for cell in cells:
9        #---------------------------------------------------------------
10     # lines 13 and 14 find all the nodes belonging to the cell and
11     # process them at a time
12     #---------------------------------------------------------------
13     nodes = AdminConfig.list('Node', cell).split(lineSeparator)
14     for node in nodes:
15        #------------------------------------------------------------
16        # lines 19-23 find all the running servers belonging to the cell
17        # and node, and process them one at a time
18        #------------------------------------------------------------
19        cname = AdminConfig.showAttribute(cell, 'name')
20        nname = AdminConfig.showAttribute(node, 'name')
21        servs = AdminControl.queryNames('type=Server,cell=' + cname +
```

```
        ',node=' + nname + ',*').split(lineSeparator)
22         print "Number of running servers on node " +
        nname + ": %s \n" % (len(servs))
23         for server in servs:
24           #-------------------------------------------------------
25           # lines 28-34 get some attributes from the server to display;
26           # invoke an operation on the server JVM to display a property.
27           #-------------------------------------------------------
28           sname = AdminControl.getAttribute(server, 'name')
29           ptype = AdminControl.getAttribute(server, 'processType')
30           pid   = AdminControl.getAttribute(server, 'pid')
31           state = AdminControl.getAttribute(server, 'state')
32            jvm = AdminControl.queryNames('type=JVM,cell=' +
        cname + ',node=' + nname + ',process=' + sname + ',*')
33           osname = AdminControl.invoke(jvm, 'getProperty', 'os.name')
34            print " " + sname + " " +  ptype + " has pid " + pid +
           "; state: " + state + "; on " +
            osname + "\n"
35
36            #-------------------------------------------------------
37            # line 40-45 find the applications running on this server and
38           # display the application name.
39           #-------------------------------------------------------
40           apps = AdminControl.queryNames('type=Application,cell=' +
       Cname + ',node=' + nname + ',process=' + sname + ',*').
          split(lineSeparator)
41           print "Number of applications running on " + sname +
           ": %s \n" % (len(apps))
42           for app in apps:
43              aname = AdminControl.getAttribute(app, 'name')
44              print aname + "\n"
45          print "-----------------------------------------------------"
46          print "\n"
```

Results

Example output:

```
Number of running servers on node mynode: 2
mynode (NodeAgent) has pid 3592; state: STARTED; on Windows 2000
Number of applications running on mynode: 0
-----------------------------------------------------

server1 (ManagedProcess) has pid 3972; state: STARTED; on Windows 2000
Number of applications running on server1: 0
-----------------------------------------------------

Number of running servers on node mynodeManager: 1
dmgr (DeploymentManager) has pid 3308; state: STARTED; on Windows 2000
Number of applications running on dmgr: 2
adminconsole
filetransfer
-----------------------------------------------------
```

Starting listener ports using scripting

These steps demonstrate how to start a listener port on an application server using scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to start a listener port on an application server. The following example returns a list of listener port MBeans:

1. Identify the listener port MBeans for the application server and assign it to the lPorts variable.
   - Using Jacl:

     ```
     set lPorts [$AdminControl queryNames type=ListenerPort,
     cell=mycell,node=mynode,process=server1,*]
     ```

   - Using Jython:

     ```
     lPorts = AdminControl.queryNames('type=ListenerPort,
     cell=mycell,node=mynode,process=server1,*')
     print lPorts
     ```

     Example output:

     ```
     WebSphere:cell=mycell,name=ListenerPort,mbeanIdentifier=server.xml#
     ListenerPort_1,type=ListenerPort,node=mynode,process=server1
     WebSphere:cell=mycell,name=listenerPort,mbeanIdentifier=ListenerPort,
     type=server.xml#ListenerPort_2,node=mynode,process=server1
     ```

2. Start the listener port if it is not started. For example:
   - Using Jacl:

     ```
     foreach lPort $lPorts {
          set state [$AdminControl getAttribute $lport started]
          if {$state == "false"} {
             $AdminControl invoke $lPort start
          }
       }
     ```

   - Using Jython:

     ```
     # get line separator
     import  java
     lineSeparator = java.lang.System.getProperty('line.separator')

     lPortsArray = lPorts.split(lineSeparator)
     for lPort in lPortsArray:
      state = AdminControl.getAttribute(lPort, 'started')
      if state == 'false':
       AdminControl.invoke(lPort, 'start')
     ```

     These pieces of Jacl and Jython code loop through the listener port MBeans. For each listener port MBean, get the attribute value for the started attribute. If the attribute value is set to `false`, then start the listener port by invoking the start operation on the MBean.

Managing generic servers using scripting

You can use WebSphere Application Server to define, start, stop, and monitor generic servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

A generic server is a server that the WebSphere Application Server manages but did not supply.

- To define a generic server, use the following example:
  – Using Jacl:

```
$AdminTask createGenericServer mynode {-name generic1 -ConfigProcDef
{{"/mydir1/myStartCommand" "arg1 arg2" "" "" "/tmp/workingDirectory"
"/mydir2/stopCommand" "argy argz"}}}
$AdminConfig save
```

– Using Jython:

```
AdminTask.createGenericServer('mynode', '[-name generic1 -ConfigProcDef
[[/mydir1/myStartCommand "a b c" "" "" /tmp/workingDirectory
/mydir2/myStopCommand "x y z"]]]')
AdminConfig.save()
```

- To start a generic server, use the `launchProcess` parameter, for example:
  – Using Jacl:

```
set nodeagent [$AdminControl queryNames *:*,type=NodeAgent]
$AdminControl invoke $nodeagent launchProcess generic1
```

  – Using Jython:

```
nodeagent = AdminControl.queryNames ('*:*,type=NodeAgent')
AdminControl.invoke(nodeagent, 'launchProcess', 'generic1')
```

  Example output:

```
true
```

  or

```
false
```

- To stop a generic server, use the `terminate` parameter, for example:
  – Using Jacl:

```
set nodeagent [$AdminControl queryNames *:*,type=NodeAgent]
$AdminControl invoke $nodeagent terminate generic1
```

  – Using Jython:

```
nodeagent = AdminControl.queryNames ('*:*,type=NodeAgent')
AdminControl.invoke(nodeagent, 'terminate', 'generic1')
```

  Example output:

```
true
```

  or

```
false
```

- To monitor the server state, use the `getProcessStatus` parameter, for example:
  – Using Jacl:

```
$AdminControl invoke $nodeagent getProcessStatus generic1
```

  Using Jython:

```
AdminControl.invoke(nodeagent, 'getProcessStatus', 'generic1')
```

  Example output:

```
RUNNING
```

  or

```
STOPPED
```

Setting development mode for server objects using scripting

You can use scripting and the wsadmin tool to configure development mode for server objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to set the development mode for a server object:

1. Locate the server object. The following example selects the first server found:
   - Using Jacl:
     ```
     set server [$AdminConfig getid /Server:server1/]
     ```
   - Using Jython:
     ```
     server = AdminConfig.getid('/Server:server1/')
     ```
2. Enable development mode:
   - Using Jacl:
     ```
     $AdminConfig modify $server "{developmentMode true}"
     ```
   - Using Jython:
     ```
     AdminConfig.modify(server, [['developmentMode', 'true']])
     ```
3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Disabling parallel startup using scripting

You can use scripting to disable parallel startup of servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to disable parallel startup:

1. Locate the server object. The following example selects the first server found:
   - Using Jacl:
     ```
     set server[$AdminConfig getid /Server:server1/]
     ```
   - Using Jython:
     ```
     server = AdminConfig.getid('/Server:server1/']
     ```
2. Enable development mode. For example:
   - Using Jacl:
     ```
     $AdminConfig modify $server "{parallelStartEnabled false}"
     ```
   - Using Jython:
     ```
     AdminConfig.modify(server, [['parallelStartEnabled', 'false']])
     ```
3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Obtaining server version information with scripting

Use the wsadmin tool and scripting to obtain server version information.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to query the server version information:

1. Identify the server and assign it to the server variable.
   - Using Jacl:

     ```
     set server [$AdminControl completeObjectName type=Server,name=server1,node=mynode,*]
     ```
   - Using Jython:

     ```
     server = AdminControl.completeObjectName('type=Server,name=server1,node=mynode,*')
     print server
     ```

   Example output:

   ```
   WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#Server_1,
   type=Server,node=mynode,process=server1,processType=ManagedProcess
   ```
2. Query the server version. The server version information is stored in the serverVersion attribute. The **getAttribute** command returns the attribute value of a single attribute, passing in the attribute name.
   - Using Jacl:

     ```
     $AdminControl getAttribute $server serverVersion
     ```
   - Using Jython:

     ```
     print AdminControl.getAttribute(server, 'serverVersion')
     ```

   Example output for a Network Deployment installation follows:

   ```
   IBM WebSphere Application Server Version Report
      ------------------------------------------------------------------------

         Platform Information
         ------------------------------------------------------------------------

                 Name: IBM WebSphere Application Server
                 Version: 5.0


         Product Information
         ------------------------------------------------------------------------

                 ID: BASE
                 Name: IBM WebSphere Application Server
                 Build Date: 9/11/02
                 Build Level: r0236.11
                 Version: 5.0.0


         Product Information
         ------------------------------------------------------------------------

                 ID: ND
                 Name: IBM WebSphere Application Server for Network Deployment
                 Build Date: 9/11/02
                 Build Level: r0236.11
                 Version: 5.0.0

      ------------------------------------------------------------------------
      End Report
      ------------------------------------------------------------------------
   ```

PropertiesBasedConfiguration command group for the AdminTask object

You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Use the following commands to manage your system configuration:

- "applyConfigProperties"
- "createPropertiesFileTemplates" on page 476
- "deleteConfigProperties" on page 476
- "extractConfigProperties" on page 477
- "validateConfigProperties" on page 480

applyConfigProperties

The **applyConfigProperties** command applies properties in a specific properties file to the configuration. The system adds attributes or configuration data to the configuration if a specific properties do not exist. If the properties exist in the configuration, the system sets the new values for the attributes.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to apply. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the applyConfigProperties command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

-validate
Specifies whether to validate the properties file before applying the changes. By default, the command validates the properties file. Specify `false` to disable validation. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.applyConfigProperties('-propertiesFileName myPropFile.props -validate true')
```

- Using Jython list:

```
AdminTask.applyConfigProperties(['-propertiesFileName', 'myPropFile.props', '-validate', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.applyConfigProperties('-interactive')
```

createPropertiesFileTemplates

The **createPropertiesFileTemplates** command creates template properties files to use to create or delete specific object types. The command stores the template properties file in the properties file specified by the propertiesFileName parameter.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file where the template is stored. (String, required)

-configType
Specifies the resource type for the template to create. (String, required)

- Specify `Server` to create a server type properties file template.
- Specify `ServerCluster` to create a server cluster type properties file template.
- Specify `Application` to create an application type properties file template.
- Specify `AuthorizationGroup` to create an authorization group type properties file template.

Optional parameters

None

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createPropertiesFileTemplates('-propertiesFileName serverTemplate.props -configType Server')
```

- Using Jython list:

```
AdminTask.createPropertiesFileTemplates(['-propertiesFileName', 'serverTemplate.props', '-configType', 'Server'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createPropertiesFileTemplates('-interactive')
```

deleteConfigProperties

The **deleteConfigProperties** command deletes properties in your configuration as designated in a properties file. The system removes the attributes or configuration data that corresponds to each property in the properties file.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to delete. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the the command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

-validate
Specifies whether to validate the properties file before applying the changes. By default, the command validates the properties file. Specify `false` to disable validation. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteConfigProperties('-propertiesFileName myPropFile.props')
```

- Using Jython list:

```
AdminTask.deleteConfigProperties(['-propertiesFileName', 'myPropFile.props'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteConfigProperties('-interactive')
```

extractConfigProperties

The **extractConfigProperties** command extracts configuration data in the form of a properties file. The system exports the most commonly used configuration data and attributes, converts the attributes to properties, and saves the data to a file. You can specify the resource of interest with the target object or the configData parameter. Use the configData parameter to specify a server, node, cluster, or application instance. If no configuration object is specified, the command extracts the profile configuration data.

Target object

Specify the object name of the configuration object of interest in the format:
`Node=nodeName:Server=serverName`

Required parameters

-propertiesFileName
Specifies the name of the properties file to extract. (String, required)

Optional parameters

-configData
Specifies the configuration object instance in the format `Node=node1`. (String, optional)

-options
Specifies additional configuration options, such as `GENERATETEMPLATE=true`. (Properties, optional)

-filterMechanism
Specifies filter information for extracting configuration properties. (String, optional)

- Specify `All` to extract all configuration properties.
- Specify `NO_SUBTYPES` to extract the properties of the given object without including the subtypes.
- Specify `SELECTED_SUBTYPES_AND_EXTENSIONS` to extract only properties of the given object type without including the subtypes. This option also prevents the command from extracting properties using extensions, if extensions exist for the object type.
- Specify `SELECTED_SUBTYPES` to extract specific configuration object subtypes specified with the selectedSubTypes parameter. This can include any subtype for a configuration object or any WCCM type that exists under the object type hierarchy.

-selectedSubTypes
Specifies the configuration properties to include or exclude when the command extracts the properties. Specify this parameter if you set the filterMechanism parameter to `NO_SUBTYPES` or `SELECTED_SUBTYPES`. The following strings are examples of sever subtypes: ApplicationServer, EJBContainer. (String, optional)

Configuration object type	Subtypes	Extensions
AdminService	None	None
Application	JDBCProvider,VariableMap	None
ApplicationServer	TransactionService, DynamicCache, WebContainer, EJBContainer, PortletContainer, SIPContainer, WebserverPluginSettings	None
AuthorizationGroup	None	None
AuthorizationTableExt	None	None
Cell	VirtualHost, DataReplicationDomain, ServerCluster, CoreGroup, NodeGroup, AuthorizationGroup, AuthorizationTableExt, Security, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, Node, VariableMap	None
CoreGroup	None	None
CoreGroupBridgeService	None	None
DynamicCache	None	None
EJBContainer	None	None
EventInfrastructureProvider	None	None
EventInfrastructureService	None	None
HAManagerService	None	None
J2CResourceAdapter	None	None
JDBCProvider	None	None

Configuration object type	Subtypes	Extensions
JMSProvider	None	None
JavaVirtualMachine	None	None
Library	None	None
MailProvider	None	None
NameServer	None	None
Node	Server, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap	The NodeMetadata Extension extracts node Metadata properties.
NodeGroup	None	None
ObjectPoolProvider	None	None
ObjectRequestBroker	None	None
PMEServerExtension	None	None
PMIModule	None	None
PMIService	None	None
PortletContainer	None	None
SIPContainer	None	None
SchedulerProvider	None	None
Security	None	None
Server	PMIService, AdminService, CoreGroupBridgeService, TPVService, ObjectRequestBroker, ApplicationServer, NameServer, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap, EventInfrastructureService, PMEServerExtension, Library, HAManagerService, PMIModule, Security	The extension lists deployed applications for a specific server.
ServerCluster	J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap	The extension lists deployed applications for a specific cluster.
TPVService	None	None
TimerManagerProvider	None	None
TransactionService	None	None
URLProvider	None	None
VariableMap	None	None
VirtualHost	None	None
WebContainer	None	None
WebserverPluginSettings	None	None
WorkManagerProvider	None	None

Return value

The command returns the name of the properties file that the system creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.extractConfigProperties('-configData Node=myNode -propertiesFileName myNodeProperties.props')
```

- Using Jython list:

```
AdminTask.extractConfigProperties(['-configData', 'Node=myNode', '-propertiesFileName', 'myNodeProperties.props'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.extractConfigProperties('-interactive')
```

validateConfigProperties

The **validateConfigProperties** command verifies that the properties in the properties file are valid and can be successfully applied to the new configuration.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to validate. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the applyConfigProperties command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

Return value

The command returns a value of `true` if the system validates the properties file.

Batch mode example usage

- Using Jython string:

```
AdminTask.validateConfigProperties('-propertiesFileName myNodeProperties.props -reportFile report.txt')
```

- Using Jython list:

```
AdminTask.validateConfigProperties(['-propertiesFileName', 'myNodeProperties.props', '-reportFile', 'report.txt'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.validateConfigProperties('-interactive')
```

NodeGroupCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the NodeGroupCommands group can be used to create and manage node groups and node group members.

The NodeGroupCommands command group for the AdminTask object includes the following commands:

addNodeGroupMember

The **addNode Group Member** command adds a member to a node group. Nodes can be members of more than one node group. The command does validity checking to ensure the following:

- Distributed and z/OS nodes are not combined in the same node group.

Target object

The target object is the node group where the member will be created. This target object is required.

Parameters and return values

-nodeName
The name of the node that you want to add to a node group. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addNodeGroupMember  WBINodeGroup {-nodeName WBINode}
```

- Using Jython string:

```
AdminTask.addNodeGroupMember ('WBINodeGroup', '[-nodeName  WBINode]')
```

- Using Jython list:

```
AdminTask.addNodeGroupMember ('WBINodeGroup', ['-nodeName',  'WBINode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addNodeGroupMember  {-interactive}
```

- Using Jython string:

```
AdminTask.addNodeGroupMember  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addNodeGroupMember  (['-interactive'])
```

createNodeGroup

The **createNode Group** command creates a new node group. A node group consists of a group of nodes that are referred to as node group members. Optionally, you can create a short name and a description for the new node group.

Target object

The node group name to be created. This target object is required.

Parameters and return values

-shortName
The short name of the node group. This parameter is optional.

-description
The description of the node group. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask createNodeGroup  WBINodeGroup
```
- Using Jython string:
```
AdminTask.createNodeGroup ('WBINodeGroup')
```
- Using Jython list:
```
AdminTask.createNodeGroup ('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask createNodeGroup  {-interactive}
```
- Using Jython string:
```
AdminTask.createNodeGroup  ('[-interactive]')
```
- Using Jython list:
```
AdminTask.createNodeGroup  (['-interactive'])
```

createNodeGroupProperty

The **createNode Group Property** command creates custom properties for a node group.

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to create. This parameter is required.

-value
The value of the custom property. This parameter is optional.

-description
The description of the custom property. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createNodeGroup Property WBINodeGroup  {-name Channel -value  "channel1"}
```

- Using Jython string:

```
AdminTask.createNodeGroup Property('WBINodeGroup',  '[-name Channel -value  channel1]')
```

- Using Jython list:

```
AdminTask.createNodeGroup Property('WBINodeGroup',  ['-name', 'Channel',  '-value', 'channel1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createNodeGroup Property {-interactive}
```

- Using Jython string:

```
AdminTask.createNodeGroup Property ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createNodeGroup Property (['-interactive'])
```

listNodeGroupProperties

The **listNode Group Properties** command displays all of the custom properties of a node group.

Target object

The target object is name of the node group. This target object is required.

Parameters and return values

- Parameters: None
- Returns: A list of all of the custom properties of a node group.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listNodeGroup Properties WBINodeGroup
```

- Using Jython string:

```
AdminTask.listNodeGroup Properties('WBINodeGroup')
```

- Using Jython list:

```
AdminTask.listNodeGroup Properties('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listNodeGroup Properties {-interactive}
```

- Using Jython string:

```
AdminTask.listNodeGroup Properties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listNodeGroup Properties (['-interactive'])
```

listNodeGroups

The **listNode Groups** command returns the list of node groups from the configuration repository. You can pass an optional node name to the command that returns the list of node groups where the node resides.

Target object

The target object is name of the node. This target object is optional.

Parameters and return values
- Parameters: None
- Returns: A list of the node groups in the cell.

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask listNodeGroups  $AdminTask listNodeGroups  nodeName
  ```
- Using Jython string:
  ```
  AdminTask.listNodeGroups   AdminTask.listNodeGroups ('nodeName')
  ```
- Using Jython list:
  ```
  AdminTask.listNodeGroups   AdminTask.listNodeGroups ('nodeName')
  ```

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask listNodeGroups  {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listNodeGroups  ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listNodeGroups  (['-interactive'])
  ```

listNodes

The **listNodes** command displays all of the nodes in the cell.

Target object

The target object is name of the node group. This target object is optional.

Parameters and return values
- Parameters: None
- Returns: A list of all the nodes in the cell

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask listNodes
  ```
- Using Jython string:
  ```
  AdminTask.listNodes()
  ```
- Using Jython list:
  ```
  AdminTask.listNodes()
  ```

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask listNodes  {-interactive}
  ```

- Using Jython string:
  ```
  AdminTask.listNodes  ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listNodes  (['-interactive'])
  ```

modifyNodeGroup

The **modify Node Group** command modifies the configuration of a node group. The node group name cannot be changed. However, its short name and description are supported. Also, its node membership can be modified.

Target object

The target object is the node group name. This target object is required.

Parameters and return values

-shortName
The short name of the node group. This parameter is optional.

-description
The description of the node group. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask modifyNodeGroup  WBINodeGroup {-shortName  WBIGroup -description  "Default node group"}
  ```
- Using Jython string:
  ```
  AdminTask.modifyNodeGroup  WBINodeGroup('[-shortName  WBIGroup -description  "WBI" node group]')
  ```
- Using Jython list:
  ```
  AdminTask.modifyNodeGroup  WBINodeGroup(['-shortName',  'WBIGroup', '-description',  "WBI", 'node', 'group'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask modifyNodeGroup  {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.modifyNodeGroup  ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.modifyNodeGroup  (['-interactive'])
  ```

modifyNodeGroupProperty

The **modify Node Group Property** command modifies custom properties for a node group

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to modify. This parameter is required.

-value
The value of the custom property. This parameter is optional.

-description
The description of the custom property. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask modifyNodeGroup Property WBINodeGroup {-name  Channel -value "channel1"}
```
- Using Jython string:
```
AdminTask.modifyNode GroupProperty('WBINodeGroup',  '[-name Channel  -value channel1]')
```
- Using Jython list:
```
AdminTask.modifyNode GroupProperty('WBINodeGroup',  ['-name', 'Channel',  '-value', 'channel1'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask modifyNodeGroup Property {-interactive}
```
- Using Jython string:
```
AdminTask.modifyNodeGroup Property ('[-interactive]')
```
- Using Jython list:
```
AdminTask.modifyNodeGroup Property (['-interactive'])
```

removeNodeGroup

The **remove Node Group** command removes the configuration of a node group. You can remove a node group if it does not contain any members. Also, the default node group cannot be removed.

Target object

The name of the node group to be removed. This target object is required.

Parameters and return values

- Parameters: None
- Returns: The node group object ID.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask removeNodeGroup  WBINodeGroup
```
- Using Jython string:
```
AdminTask.removeNodeGroup ('WBINodeGroup')
```
- Using Jython list:
```
AdminTask.removeNodeGroup ('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask removeNodeGroup  {-interactive}
```
- Using Jython string:

```
AdminTask.removeNodeGroup  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup  (['-interactive'])
```

removeNodeGroupMember

The **removeNode Group Member** command removes the configuration of a node group member.

- A node must always be a member of at least one node group.
- You cannot remove a node from a node group that is part of a cluster in that node group.

Target object

The target object is the node group containing the member to be removed. This target object is required.

Parameters and return values

-nodeName
The name of the node to remove from a node group. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeNode GroupMember WBINodeGroup  {-nodeName WBINode}
```

- Using Jython string:

```
AdminTask.removeNode GroupMember('WBINodeGroup',  '[-nodeName WBINode]')
```

- Using Jython list:

```
AdminTask.removeNode GroupMember('WBINode Group', ['-nodeName',  'WBINode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeNodeGroup Member {-interactive}
```

- Using Jython string:

```
AdminTask.removeNodeGroup Member ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup Member (['-interactive'])
```

removeNodeGroupProperty

The **removeNode Group Property** command removes custom properties of a node group.

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to remove. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

  ```
  $AdminTask removeNodeGroup Property WBINodeGroup  {-name Channel}
  ```

- Using Jython string:

  ```
  AdminTask.removeNodeGroup Property('WBINodeGroup',  '[-name Channel]')
  ```

- Using Jython list:

  ```
  AdminTask.removeNodeGroup Property('WBINodeGroup',  ['-name', 'Channel'])
  ```

Interactive mode example usage:

- Using Jacl:

  ```
  $AdminTask removeNodeGroup Property {-interactive}
  ```

- Using Jython string:

  ```
  AdminTask.removeNodeGroup Property ('[-interactive]')
  ```

- Using Jython list:

  ```
  AdminTask.removeNodeGroup Property (['-interactive'])
  ```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Utility command group of the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the Utility group can be used to change the host name of a node, to query for the name of the deployment manager, and to determine if the system is a single server or network deployment.

The Utility command group for the AdminTask object includes the following commands:

- "changeHostName"
- "getDmgrProperties" on page 489
- "isFederated" on page 489

changeHostName

Use the **changeHost Name** command to change the host name of a node.

Target object

None

Parameters and return values

-hostName
The new host name. (String, required)

-nodeName
The name of the node whose host name will be changed. (String, required)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask changeHostName {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.changeHostName ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.changeHostName (['-interactive'])
  ```

getDmgrProperties

Use the **getDmgr Properties** command to return the name of the deployment manager.

Target object

None

Parameters and return values
- Parameters: None
- Returns: The name of the deployment manager in a network deployment system. Returns an empty string if the system is a single server.

Examples

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask getDmgrProperties {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getDmgrProperties ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getDmgrProperties (['-interactive'])
  ```

isFederated

Use the **isFederated** command to check if the system is a single server or network deployment.

Target object

None

Parameters and return values
- Parameters: None
- Returns: Boolean. `true` if the system is a network deployment system. Otherwise it returns `false`.

Examples

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask isFederated {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.isFederated ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.isFederated (['-interactive'])
  ```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

ManagedObjectMetadata command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the ManagedObjectMetadata group can be used to retrieve configuration and metadata information for a specified node.

The ManagedObjectMetadata command group for the AdminTask object includes the following commands:

- “compareNodeVersion”
- “getMetadataProperties” on page 491
- “getMetadataProperty” on page 491
- “getNodeBaseProductVersion” on page 492
- “getNodeMajorVersion” on page 493
- “getNodeMinorVersion” on page 493
- “getNodePlatformOS” on page 494
- “getNodeSysplexName” on page 494
- “isNodeZOS” on page 495

compareNodeVersion

The **compareNode Version** command compares the WebSphere Application Server version given a node that you specify and an input version.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

-version
A version number that you want to compare to the WebSphere Application Server version number.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask compareNode Version {-nodeName node1 -version 5}
  ```
- Using Jython string:
  ```
  AdminTask.compareNode Version('[-nodeName node1 -version 5]')
  ```
- Using Jython list:
  ```
  AdminTask.compareNode Version(['-nodeName', 'node1', '-version', '5'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask compareNode Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.compareNode Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.compareNode Version (['-interactive'])
  ```

getMetadataProperties

The **getMetadata Properties** command obtains all metadata for the node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask getMetadata Properties {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getMetadata Properties('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getMetadata Properties(['-nodeName', 'node1'])
  ```

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask getMetadataProperties {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getMetadataProperties ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getMetadataProperties (['-interactive'])
  ```

getMetadataProperty

The **getMetadata Property** command obtains metadata with the specified key for the node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

-propertyName
Metadata property key.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getMetadataProperty {-nodeName node1 -property Name com.ibm.websphere.base ProductVersion}
```

- Using Jython string:

```
AdminTask.getMetadataProperty ('[-nodeName node1 -propertyName com.ibm. websphere.baseProductVersion]')
```

- Using Jython list:

```
AdminTask.getMetadataProperty (['-nodeName', 'node1', '-propertyName', 'com.ibm. websphere.baseProductVersion'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getMetadataProperty {-interactive}
```

- Using Jython string:

```
AdminTask.getMetadataProperty ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getMetadataProperty (['-interactive'])
```

getNodeBaseProductVersion

The **getNode Base Product Version** command returns the version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getNodeBaseProduct Version {-nodeName node1}
```

- Using Jython string:

```
AdminTask.getNodeBaseProduct Version('[-nodeName node1]')
```

- Using Jython list:

```
AdminTask.getNodeBaseProduct Version(['-nodeName', 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getNodeBaseProduct Version {-interactive}
```

- Using Jython string:

```
AdminTask.getNodeBaseProduct Version ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getNodeBaseProduct Version (['-interactive'])
```

getNodeMajorVersion

The **getNode Major Version** command returns the major version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMajorVersion {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMajorVersion ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMajorVersion (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMajor Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMajor Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMajor Version (['-interactive'])
  ```

getNodeMinorVersion

The **getNode Minor Version** command returns the minor version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMinorVersion {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMinorVersion ('[-nodeName node1]')
  ```

- Using Jython list:
  ```
  AdminTask.getNodeMinorVersion (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMinor Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMinor Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMinor Version (['-interactive'])
  ```

getNodePlatformOS

The **getNode Platform OS** command returns the operating system name for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodePlatformOS {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodePlatformOS ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodePlatformOS (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodePlatformOS {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodePlatformOS ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodePlatformOS (['-interactive'])
  ```

getNodeSysplexName

The **getNode Sysplex Name** command returns the sysplex name for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeSysplexName {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeSysplexName ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeSysplexName (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeSysplexName {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeSysplexName ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeSysplexName (['-interactive'])
  ```

isNodeZOS

The **isNodeZOS** command tests if a node that you specify is running on the z/OS platform. This command does not apply to distributed platforms or to WebSphere Application Server-Express.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask isNodeZOS {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.isNodeZOS('[ -nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.isNodeZOS([' -nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask isNodeZOS {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.isNodeZOS ('[-interactive]')
  ```
- Using Jython list:

```
AdminTask.isNodeZOS (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

ServerManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the ServerManagement group can be used to create and manage application server, Web server, proxy server, generic server and Java virtual machine (JVM) configurations.

The ServerManagement command group for the AdminTask object includes the following commands:

• “showTemplateInfo” on page 523

createApplicationServer

Use the **createApplicationServer** command to create a new application server.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server that you want to create. (String, required)

Optional parameters

-templateName
The name of the template from which to base the server. (String, optional)

-genUniquePorts
Specifies that unique ports should be created for the server. (Boolean, optional)

-templateLocation
The configuration Id that represents the location of a template. Specify the `_Websphere_Config_Data_Id=templates/servertypes/APPLICATION_SERV ER|servertype-metadata.xml` configuration Id to create an application server. (String, optional)

Examples

Batch mode example usage:

• Using Jacl:

```
$AdminTask createApplicationServer ndnode1 {-name test1 -templateName default}
```

• Using Jython string:

```
AdminTask.createApplicationServer(ndnode1, '[-name test1 -templateName
 default]")
```

• Using Jython list:

```
AdminTask.createApplicationServer(ndnode1, ['-name', 'test1',  '-templateName',
 'default'])
```

Interactive mode example usage:

• Using Jacl:

```
$AdminTask createApplicationServer {-interactive}
```

• Using Jython:

```
AdminTask.createApplicationServer ('-interactive')
```

createApplicationServerTemplate

The **createApplicationServerTemplate** command creates a new application server template.

Target object

None

Required parameters

-templateName
Specifies the name of the application server template that you want to create. (String, required)

-serverName
Specifies the name of the server from which to base the template. (String, required)

-nodeName
Specifies the node that corresponds to the server from which to base the template. (String, required)

Optional parameters

-description
Specifies the description of the template. (String)

-templateLocation
Specifies a configuration Id that represents the location to place the template. (String)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=configuration_id
```

The configuration Id can be one of the following values:

- To create a Web server template:
  `templates\servertypes\WEB_SERVER|servertype-metadata.xml`
- To create an application server template:
  `templates\servertypes\APPLICATION_SERVER|servertype-metadata.xml`
- To create a generic server template:
  `templates\servertypes\GENERIC_SERVER|servertype-metadata.xml`
- To create a proxy server template:
  `templates\servertypes\PROXY_SERVER|servertype-metadata.xml`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServerTemplate {-templateName newTemplate -serverName
 server1  -nodeName ndnode1 -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=APPLICATION_SERVER,_Websphere_Config_Data_Id=
 templates/servertypes/APPLICATION_SERVER|servertype-metadata.xml}
```

- Using Jython string:

```
AdminTask.createApplicationServerTemplate('[-templateName newTemplate -serverName
 server1 -nodeName ndnode1  -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=
 APPLICATION_SERVER,_Websphere_Config_Data_Id=templates/servertypes/APPLICATION_SERVER|servertype-metadata.xml]')
```

- Using Jython list:

```
AdminTask.createApplicationServerTemplate(['-templateName',  'newTemplate', '-serverName',
 'server1', '-nodeName',  'ndnode1', '-description', "This is my new template"])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.createApplicationServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createApplicationServerTemplate (['-interactive'])
```

createGenericServer

Use the **createGenericServer** command to create a new generic server in the configuration. A generic server is a server that the WebSphere Application Server manages, but did not supply. The **createGenericServer** command provides an additional step, ConfigProcDef, that you can use to configure the parameters that are specific to generic servers.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server that you want to create.

Optional parameters

-templateName
Picks up a server template. This step provides a list of application server templates for the node and server type. The default value is the default templates for the server type. (String, optional)

-genUniquePorts
The port for the server. (Integer, optional)

-templateLocation
The configuration Id that represents the location of a template. Specify the `_Websphere_Config_Data_Id=templates/servertypes/GENERIC_SERVER|servertype-metadata.xml` configuration Id to create a generic server. (ObjectName)

-startCommand
Indicates the path to the command that will run when this generic server is started. (String, optional)

-startCommandArgs
Indicates the arguments to pass to the startCommand when the generic server is started. (String, optional)

-executableTargetKind
Specifies whether a Java class name (use `JAVA_CLASS`) or the name of an executable JAR file (use `EXECUTABLE_JAR`) will be used as the executable target for this process. This field should be left blank for binary executables. This parameter is only applicable for Java processes. (String optional)

-executableTarget
Specifies the name of the executable target (a Java class containing a main() method or the name of an executable JAR), depending on the executable target type. This field should be left blank for binary executables. This parameter is only applicable for Java processes. (String, optional)

-workingDirectory
Specifies the working directory for the generic server.

-stopCommand
Indicates the path to the command that will run when this generic server is stopped. (String, optional)

-stopCommandArgs
Indicates the arguments to pass to the stopCommand parameter when the generic server is stopped. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createGenericServer jim667BaseNode {-name jgeneric  -ConfigProcDef
 {{"/usr/bin/myStartCommand" "arg1 arg2" "" ""  "/tmp/workingDirectory" "/tmp/stopCommand" "argy argz"}}}
```

- Using Jython string:

```
AdminTask.createGenericServer('jim667BaseNode', '[-name jgeneric  -ConfigProcDef
 [[/usr/bin/myStartCommand "arg1 arg2" "" ""  /tmp/workingDirectory /tmp/StopCommand "argy argz"]]]')
```

- Using Jython list:

```
AdminTask.createGenericServer('jim667BaseNode', ['-name', 'jgeneric',
 '-ConfigProcDef',  [['/usr/bin/myStartCommand', "arg1 arg2" "" "", '/tmp/workingDirectory', '/tmp/StopCommand', "argy
 argz"]]])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGenericServer {-interactive}
```

- Using Jython string:

```
AdminTask.createGenericServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createGenericServer (['-interactive'])
```

createGenericServerTemplate

Use the createGenericServerTemplate command to create a server template based on a server configuration.

Target object

None.

Required parameters

-serverName
Specifies the name of the server of interest. (String, required)

-nodeName
Specifies the name of the node of interest. (String, required)

-templateName
Specifies the name of the template to create. (String, required)

Optional parameters

-description
Provides a description for the template to be created. (String, optional)

-templateLocation
Specifies a configuration Id that represents the location of the template. If this parameter is not specified, the system uses the default location. (String, optional)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=configuration_id
```

The configuration Id can be one of the following values:

- To create a Web server template:
  `templates\servertypes\WEB_SERVER|servertype-metadata.xml`
- To create an application server template:
  `templates\servertypes\APPLICATION_SERVER|servertype-metadata.xml`
- To create a generic server template:
  `templates\servertypes\GENERIC_SERVER|servertype-metadata.xml`
- To create a proxy server template:

```
templates\servertypes\PROXY_SERVER|servertype-metadata.xml
```

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGenericServerTemplate {-interactive}
```

- Using Jython:

```
AdminTask.createGenericServerTemplate('-interactive')
```

createProxyServer

Use the **createProxyServer** command to create a new proxy server in the configuration. The proxy server is a specific type of application server that routes HTTP requests to content servers that perform the work. The proxy server is the initial point of entry, after the firewall, for requests into the enterprise.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server to create. (String)

Optional parameters

-templateName
Picks up a server template. This step provides a list of application server templates for the node and server type. The default value is the default templates for the server type. (String)

-genUniquePorts
Specifies whether the system generates unique HTTP ports for the server. The default value is `true`. Specify `false` if you do not want to generate unique HTTP ports for the server. (Boolean)

-templateLocation
Specifies the location of the template on your system. Use the system defined location if you are unsure of the location. (String)

Specifies the specific short name of the server. Each server should have a specific short name. The value of this parameter must be 8 uppercase characters or less. If you do not specify a value for the specificShortName parameter, the system generates a unique short name.

Specifies the generic short name of the server. Each member of a cluster shares the same generic short name. Assign a unique generic short name to servers that do not belong to a cluster. The value of this parameter must be 8 uppercase characters or less. If you do not specify a value for the genericShortName parameter, the system generates a unique short name. (String)

-clusterName
Specifies the name of the cluster to which the system assigns the server. (String)

Optional steps

-ConfigCoreGroup

coregroupName
Specifies the name of the core group to configure. (String)

-selectProtocols

list Specifies a list of protocols that the proxy server supports. (java.util.List)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createProxyServer myNode {-name myProxyServer -templateName myTemplate
 -ConfigCoreGroup [-coregroupName [myCoreGroup]] -selectProtocols [-list [HTTP, SIP]]}
```

- Using Jython string:

```
AdminTask.createProxyServer('myNode', '[-name myProxyServer -templateName myTemplate
 -ConfigCoreGroup [-coregroupName [myCoreGroup]] -selectProtocols [-list [HTTP, SIP]]]')
```

- Using Jython list:

```
AdminTask.createProxyServer(myNode, ['-name', 'myProxyServer', '-templateName', 'myTemplate',
 '-ConfigCoreGroup', '[-coregroupName [myCoreGroup]]', '-selectProtocols', '[-list [HTTP, SIP]]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createProxyServer {-interactive}
```

- Using Jython:

```
AdminTask.createProxyServer('-interactive')
```

createProxyServerTemplate

Use the **createProxyServerTemplate** command to create a new proxy server template based on an existing proxy server configuration.

Target object

None

Required parameters

-templateName
Specifies the name of the proxy server template to create. (String)

-serverName
Specifies the name of the proxy server of interest. (String)

-nodeName
Specifies the name of the node of interest. (String)

Optional parameters

-description
Specifies a description for the new server template. (String)

-templateLocation
Specifies the location of the template on your system. Use the system defined location if you are unsure of the location. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createProxyServerTemplate {-templateName proxyServerTemplate -serverName proxyServer1
 -nodeName myNode}
```

- Using Jython string:

```
AdminTask.createProxyServerTemplate('[-templateName proxyServerTemplate -serverName proxyServer1
 -nodeName myNode]')
```

- Using Jython list:

```
AdminTask.createProxyServerTemplate(['-templateName', 'proxyServerTemplate', '-serverName',
 'proxyServer1', '-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createProxyServerTemplate {-interactive}
```

- Using Jython:

```
AdminTask.createProxyServerTemplate('[-interactive]')
```

createServerTemplate

Use the createServerTemplate command to create a server using a template.

Target object

None.

Required parameters

-serverName
Specifies the name of the server of interest. (String, required)

-nodeName
Specifies the name of the node of interest. (String, required)

-templateName
Specifies the name of the template to create. (String, required)

Optional parameters

-description
Provides a description for the template. (String, optional)

-templateLocation
Specifies a configuration Id that represents the location of the template. If this parameter is not specified, the system uses the default location. (String, optional)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=configuration_id
```

The configuration Id can be one of the following values:

- To create a Web server template:
  `templates\servertypes\WEB_SERVER|servertype-metadata.xml`
- To create an application server template:
  `templates\servertypes\APPLICATION_SERVER|servertype-metadata.xml`
- To create a generic server template:
  `templates\servertypes\GENERIC_SERVER|servertype-metadata.xml`
- To create a proxy server template:
  `templates\servertypes\PROXY_SERVER|servertype-metadata.xml`

Sample output

The command returns the object name of the template that was created.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createServerTemplate {-interactive}
```

- Using Jython:

```
AdminTask.createServerTemplate('-interactive')
```

createServerType

Use the createServerType command to define a server type.

Target object

None.

Required parameters

-version
Specifies the product version. (String, required)

-serverType
Specifies the server type of interest. (String, required)

-createTemplateCommand
Specifies the command to use to create a server template. (String, required)

-createCommand
Specifies the command to use to create a server. (String, required)

-configValidator
Specifies the name of the configuration validator class. (String, required)

Optional parameters

-defaultTemplateName
Specifies the name of the default template. The default value is `default`. (String, optional)

Sample output

The command returns the object name of the server type that was created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createServerType {-version version -serverType serverType
 -createTemplateCommand name -createCommand name}
```

- Using Jython string:

```
AdminTask.createServerType('[-version version -serverType serverType
 -createTemplateCommand name -createCommand name]')
```

- Using Jython list:

```
AdminTask.createServerType(['-version', 'version', '-serverType',
 'serverType','-createTemplateCommand', 'name', '-createCommand', 'name'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createServerType {-interactive}
```

- Using Jython:

```
AdminTask.createServerType('-interactive')
```

createWebServer

Use the **createWebServer** command to create a Web server definition. This command creates a Web server definition using a template and configures the Web server definition properties. Web server definitions generate and propagate the `plugin-config.xml` file for each Web server. For IBM HTTP Server only, you can use Web server definitions to administer and configure IBM HTTP Server Web servers with the administrative console. These functions include: Start, Stop, View logs, View and Edit configuration files.

Target object

Specifies the name of the node (String, required).

Required parameters

-name
Specifies the name of the server. (String, required)

-serverConfig
Specifies the Web server definition properties. Use this parameter and associated options to specify configuration properties for the IBM HTTP Server. Specify the following values in order in a list with the -serverConfig parameter:

webPort
Specifies the port number of the Web server. This option is required for all Web servers. (Integer, required)

webInstallRoot
Specifies the install path directory for the Web server. This option is required for IBM HTTP Server Admin Function. (String, required)

pluginInstallRoot
Specifies the installation root directory where the plug-in for the Web server is installed. This option is required for all Web servers. (String, required)

configurationFile
Specifies the file path for the IBM HTTP Server. This option is required for view and edit of the IBM HTTP Server Configuration file only. (String, required)

serviceName
Specifies the windows service name on which to start the IBM HTTP Server. This option is required for start and stop of the IBM HTTP Server Web server only. (String, required)

errorLogfile
Specifies the path for the IBM HTTP Server error log (`error.log`) (String, required)

accessLogfile
Specifies the path for the IBM HTTP Server access log (access.log). (String, required)

webProtocol
Specifies the IBM HTTP Server administration server running with an unmanaged or remote Web server. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)

webAppMapping
Specifies configuration information for Web application mapping. (String, required)

-remoteServerConfig
Specifies additional Web server definition properties that are only necessary if the IBM HTTP Server Web server is installed on a machine remote from the application server. Specify the following values in order in a list with the remoteServerConfig parameter:

adminPort
Specifies the port of the IBM HTTP Server administrative server. The administration server is installed on the same machine as the IBM HTTP Server and handles administrative requests to the IBM HTTP Server Web server. (String, required)

adminProtocol
Specifies the administrative protocol title. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)

adminUserID
Specifies the user ID, if authentication is activated on the Administration server in the admin configuration file (`admin.conf`). This value should match the authentication in the admin.conf file. (String, optional)

adminPasswd
Specifies the password for the user ID. The password is generated by the htpasswd utility in the `admin.passwd` file. (String, optional)

Optional parameters

-templateName
Specifies the name of the template that you want to use. Templates include the following: `IHS`, `iPlanet`, `IIS`, `DOMINO`, `APACHE`. The default template is `IHS`. (String, required)

-genUniquePorts
Indicates that you want to generate unique ports. (Boolean, optional)

-templateLocation
The configuration Id that represents the location of a template. Specify the `_Websphere_Config_Data_Id=templates/servertypes/WEB_SERVER|servertype-metadata.xml` configuration Id to create a generic server. (ObjectName)

-clusterName
Specifies the cluster of interest. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWebServer myNode {-name wsname -serverConfig {{80
 /opt/path/to/ihs   /opt/path/to/plugin  /opt/path/to/plugin.xml  "windows service"
 /opt/path/to/error.log   /opt/path/to/access.log  HTTP}} -remoteServerConfig {{8008 user
 password HTTP}}
```

- Using Jython list:

```
print AdminTask.createWebServer(myNode, ['-name', wsname,
 '-serverConfig',[['80','/opt/path/to/ihs', '/opt/path/to/plugin','/opt/path/to/plugin.xml',
 'windows service','/opt/path/to/error.log', '/opt/path/to/access.log','HTTP']],'-remoteServerConfig',
 [['8008','user','password','HTTP']]])
```

- Using Jython string:

```
AdminTask.createWebServer('myNode','-name wsname -serverConfig [80
 /opt/path/to/ihs  /opt/path/to/plugin /opt/path/to/plugin.xml "windows service" /opt/path/to/error.log /opt/path/to/access.log
 HTTP] -remoteServerConfig [8008 user password HTTP]')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWebServer -interactive
```

- Using Jython string:

```
AdminTask.createWebServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createWebServer (['-interactive'])
```

deleteServer

Use the **deleteServer** command to delete a server.

Target object

None

Required parameters

-serverName
The name of the server to delete. (String, required)

-nodeName
The name of the node for the server that you want to delete. (String, required)

Optional parameters

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteServer {-serverName server_name -nodeName node_name}
```

- Using Jython string:

```
AdminTask.deleteServer('[-serverName server_name -nodeName node_name]')
```

- Using Jython list:

```
AdminTask.deleteServer(['-serverName', 'server_name', '-nodeName',
 'node_name'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteServer {-interactive}
```

- Using Jython string:

```
AdminTask.deleteServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteServer (['-interactive'])
```

deleteServerTemplate

Use the **deleteServerTemplate** command to delete a server template. You cannot delete templates that are defined by the system. You can only delete server templates that you created. This command deletes the directory that hosts the server template.

Target object

The name of the template to delete. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteServerTemplate
 template_name(templates/serverTypes/APPLICATION_SERVER/servers /newTemplate|server.xml#Server_1105015708079)
```

- Using Jython string:

```
AdminTask.deleteServerTemplate('template_name(templates/serverTypes/APPLICATION_SERVER/servers
 /newTemplate|server.xml#Server_1105015708079)')
```

- Using Jython list:

```
AdminTask.deleteServerTemplate('template_name(templates/serverTypes/APPLICATION_SERVER/servers
 /newTemplate|server.xml#Server_1105015708079)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.deleteServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteServerTemplate (['-interactive'])
```

getJavaHome

Use the **getJavaHome** command to get the Java home value.

Target object

None.

Required parameters

-serverName
Specifies the name of the server. (String, required)

-nodeName
Specifies the name of the node. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getJavaHome {-nodeName mynode -serverName myserver}
```

- Using Jython string:

```
AdminTask.getJavaHome ('[-nodeName mynode -serverName myserver]')
```

- Using Jython list:

```
AdminTask.getJavaHome (['-nodeName' 'mynode' '-serverName' 'myserver'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getJavaHome {-interactive}
```

- Using Jython string:

```
AdminTask.getJavaHome ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getJavaHome (['-interactive'])
```

getServerType

The **getServerType** command returns the type of the server that you specify.

Target object

None

Optional parameters

-serverName
The name of the server. (String)

-nodeName
The name of the node. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getServerType {-serverName test2 -nodeName ndnode1}
```

- Using Jython string:

```
AdminTask.getServerType('[-serverName test2 -nodeName ndnode1]')
```

- Using Jython list:

```
AdminTask.getServerType(['-serverName', 'test2', '-nodeName', 'ndnode1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getServerType {-interactive}
```

- Using Jython string:

```
AdminTask.getServerType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getServerType (['-interactive'])
```

listServers

The **listServers** command returns a list of servers.

Target object

None

Optional parameters

serverType
Specifies the type of the server. Used to filter the results. (String, optional)

nodeName
Specifies the name of the node. Used to filter the results. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServers {-serverType APPLICATION_SERVER -nodeName ndnode1}
```

- Using Jython string:

```
AdminTask.listServers('[-serverType APPLICATION_SERVER -nodeName ndnode1]')
```

- Using Jython list:

```
AdminTask.listServers(['-serverType', 'APPLICATION_SERVER',  '-nodeName',
 'ndnode1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServers {-interactive}
```

- Using Jython string:

```
AdminTask.listServers ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServers (['-interactive'])
```

listServerTemplates

Use the **listServerTemplates** command to list server templates.

Target object

None

Optional parameters

-version
The version of the template that you want to list. (String, optional)

-serverType
Specify this option if you want to list templates for a specific server type. (String, optional)

-name
Specify this option to look for a specific template. (String, optional)

-queryExp
A key and value pair that you can use to find templates by properties. For example, `com.ibm.websphere.nodeOperatingSystem=os390` (String[], optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServerTemplates {-version 6.0.0.0 -serverType
 APPLICATION_SERVER}
```

- Using Jython string:

```
AdminTask.listServerTemplates('[-version 6.0.0.0 -serverType
 APPLICATION_SERVER]')
```

- Using Jython list:

```
AdminTask.listServerTemplates(['-version', '6.0.0.0',  '-serverType',
 'APPLICATION_SERVER'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServerTemplates {-interactive}
```

- Using Jython string:

```
AdminTask.listServerTemplates ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServerTemplates (['-interactive'])
```

listServerTypes

Use the **listServerTypes** command to display all the current server types. For example, `APPLICATION_SERVER, WEB_SERVER, GENERIC_SERVER`

Target object

The node name for which you want to list the valid types. For example, the types that are only valid on z/OS will appear on a z/OS node. (String, optional)

Parameters and return values

- Parameters: None
- Returns: A list of server types that you can define on a node. If you do not specify the target object, this command returns all of the server types defined in the entire cell.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServerTypes ndnode1
```

- Using Jython string:

```
AdminTask.listServerTypes(ndnode1)
```

- Using Jython list:

```
AdminTask.listServerTypes(ndnode1)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServerTypes {-interactive}
```

- Using Jython string:

```
AdminTask.listServerTypes ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServerTypes (['-interactive'])
```

setJVMDebugMode

Use the **setJVMDebugMode** command to set the Java virtual machine (JVM) debug mode for the application server.

Target object

None

Required parameters

-serverName
The name of the server whose JVM properties will be modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-debugMode
Specifies whether to run the JVM in debug mode. The default is not to enable debug mode. (Boolean, required)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMDebugMode {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMDebugMode ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMDebugMode (['-interactive'])
```

setGenericJVMArguments

Use the **setGenericJVMArguments** command passes command line arguments to the Java virtual machine (JVM) code that starts the application server process.

Target object

None

Required parameters

-serverName
Specifies the name of the server that contains the JVM properties that are modified. If only one server exists in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-genericJvmArguments
Specifies that the command line arguments pass to the Java virtual machine code that starts the application server process. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setGenericJVMArguments {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setGenericJVMArguments('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setGenericJVMArguments(['-serverName', 'server1',  '-nodeName',
 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setGenericJVMArguments {-interactive}
```

- Using Jython string:

```
AdminTask.setGenericJVMArguments ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setGenericJVMArguments (['-interactive'])
```

setJVMInitialHeapSize

Use the **setJVMInitialHeapSize** command to set the Java Virtual Machine (JVM) initial heap size for the application server.

Target object

None

Parameters and return values

-serverName
The name of the server whose JVM properties are modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-initialHeapSize
Specifies the initial heap size available to the JVM code, in megabytes. (Integer, required)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMInitialHeapSize {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMInitialHeapSize ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMInitialHeapSize (['-interactive'])
```

setJVMMaxHeapSize

Use the **setJVMMaxHeapSize** command to set the Java virtual machine (JVM) maximum heap size for the application server.

Target object

None

Parameters and return values

-serverName
The name of the server whose JVM properties are modified. If there is only one server in the configuration, (String, required)

-nodeName
The node name where the server locates. If the server name is unique in the cell, this parameter is optional. (String, required)

-maximumHeapSize
Specifies the maximum heap size available to the JVM code, in megabytes. (Integer, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMMaxHeapSize {-serverName server1 -nodeName node1 -maximumHeapSize
 10}
```

- Using Jython string:

```
AdminTask.setJVMMaxHeapSize('[-serverName server1 -nodeName node1 -maximumHeapSize
 10]')
```

- Using Jython list:

```
AdminTask.setJVMMaxHeapSize(['-serverName', 'server1', '-nodeName', 'node1',
 '-maximumHeapSize', '10'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMMaxHeapSize {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMMaxHeapSize ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMMaxHeapSize (['-interactive'])
```

setJVMMode

Use the **setJVMMode** command to set the Java virtual machine mode.

Target object

None.

Parameters and return values

-serverName
The name of the server. (String, required)

-nodeName
The name of the node. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMMode {-nodeName myNode -serverName myserver -mode
 64bit}
```

- Using Jython string:

```
AdminTask.setJVMMode ('[-nodeName myNode -serverName myserver -mode
 64bit]')
```

- Using Jython list:

```
AdminTask.setJVMMode (['-nodeName', 'myNode', '-serverName', 'myserver', '-mode',
 '64bit'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMMode {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMMode ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMMode (['-interactive'])
```

setJVMProperties

Use the **setJVMProperties** command to set the Java virtual machine (JVM) configuration for the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server for which the JVM properties will be modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the entire cell, this parameter is optional. (String, required)

Optional parameters

-classpath
Specifies the standard class path in which the Java virtual machine (JVM) code looks for classes. (String, optional)

-bootClasspath
Bootstrap classes and resources for JVM code. This option is only available for JVM instructions that support bootstrap classes and resources. You can separate multiple paths by a colon (:) or semi-colon (;), depending on the operating system of the node. (String, optional)

-verboseModeClass
Specifies whether to use verbose debug output for class loading. The default is not to enable verbose class loading. (Boolean, optional)

-verboseModeGarbageCollection
Specifies whether to use verbose debug output for garbage collection. The default is not to enable verbose garbage collection. (Boolean, optional)

-verboseModeJNI
Specifies whether to use verbose debug output for native method invocation. The default is not to enable verbose Java Native Interface (JNI) activity. (Boolean, optional)

-initialHeapSize
Specifies the initial heap size in megabytes that is available to the JVM code. (Integer, optional)

-maximumHeapSize
Specifies the maximum heap size available in megabytes to the JVM code. (Integer, optional)

-runHProf
This parameter only applies to WebSphere Application Server version. It specifies whether to use HProf profiler support. To use another profiler, specify the custom profiler settings using the hprofArguments parameter. The default is not to enable HProf profiler support. (Boolean, optional)

-hprofArguments
This parameter only applies to WebSphere Application Server version. It specifies command-line profiler arguments to pass to the JVM code that starts the application server process. You can specify arguments when HProf profiler support is enabled. (String, optional)

-debugMode
Specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. (Boolean, optional)

-debugArgs
Specifies the command line debug arguments to pass to the JVM code that starts the application server process. You can specify arguments when the debug mode is enabled. (String, optional)

-genericJvmArguments
Specifies the command line arguments to pass to the JVM code that starts the application server process. (String, optional)

-executableJarFileName
Specifies a full path name for an executable JAR file that the JVM code uses. (String, optional)

-disableJIT
Specifies whether to disable the just in time (JIT) compiler option of the JVM code. (Boolean, optional)

-osName
Specifies the JVM settings for a given operating system. When started, the process uses the JVM settings for the operating system of the node. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMProperties {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setJVMProperties('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setJVMProperties(['-serverName', 'server1', '-nodeName', 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMProperties {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMProperties (['-interactive'])
```

setJVMSystemProperties

Use the **setJVMSystemProperties** command to set the Java virtual machine (JVM) system property for the process of the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose JVM system properties will be set. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-propertyName
Specifies the property name. (String, required)

-propertyValue
Specifies the property value. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMSystemProperties {-serverName server1 -nodeName node1
 -propertyName  test.property -propertyValue testValue}
```

- Using Jython string:

```
AdminTask.setJVMSystemProperties('[-serverName server1 -nodeName node1
 -propertyName  test.property -propertyValue testValue]')
```

- Using Jython list:

```
AdminTask.setJVMSystemProperties(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName', 'test.property',  '-propertyValue', 'testValue'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMSystemProperties {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMSystemProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMSystemProperties (['-interactive'])
```

setProcessDefinition

Use the **setProcessDefinition** command to set the process definition of an application server.

Target object

None

Required parameters

-serverName
The name of the server for which you want to modify the process definition. If there is only one server in the entire configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the entire cell, this parameter is optional. (String, required)

Optional parameters

-executableName
Specifies the executable name that is invoked to start the process. This parameter is only applicable to WebSphere Application Server version. (String, optional)

-executableArguments
Specifies the arguments that are passed to the process when it is started. This parameter is only applicable to WebSphere Application Server version. (String, optional)

-workingDirectory
Specifies the file system directory that the process uses for the current working directory. (String, optional)

-executableTargetKind
Specifies the type of the executable target. Valid values include JAVA_CLASS and EXECUTABLE JAR. (String, optional)

-executableTarget
Specifies the name of the executable target. The executable target is a Java class containing a main() method, or the name of an executable JAR file. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setProcessDefinition {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setProcessDefinition('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setProcessDefinition(['-serverName', 'server1', '-nodeName',
 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setProcessDefinition {-interactive}
```

- Using Jython string:

```
AdminTask.setProcessDefinition ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setProcessDefinition (['-interactive'])
```

setTraceSpecification

Use the **setTraceSpecification** command to set the trace specification for the server. If the server is running new trace specification the change takes effect immediately. This command also saves the trace specification in configuration.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose trace specification will be set. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-traceSpecification
Specifies the trace specification. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setTraceSpecification {-serverName server1 -nodeName node1
 -traceSpecification  com.ibm.*=all=enabled}
```

- Using Jython string:

```
AdminTask.setTraceSpecification('[-serverName server1 -nodeName node1
 -traceSpecification  com.ibm.*=all=enabled]')
```

- Using Jython list:

```
AdminTask.setTraceSpecification(['-serverName', 'server1',  '-nodeName', 'node1',
 '-traceSpecification',  'com.ibm.*=all=enabled'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setTraceSpecification {-interactive}
```

- Using Jython string:

```
AdminTask.setTraceSpecification ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setTraceSpecification (['-interactive'])
```

showJVMProperties

Use the **showJVMProperties** command to list the Java virtual machine (JVM) configuration for the server of the application process.

Target object

None

Required parameters

-serverName
Specifies the name of the Server whose JVM properties are shown. If there is only one server in the entire configuration, then this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server locates. If the server name is unique in the entire cell, then this parameter is optional. (String, required)

-propertyName
If you specify this parameter, the value of this property is returned. If you do not specify this parameter, all JVM properties will return in list format. Each element in the list is a property name and value pair. (String, optional)

Optional parameters

-propertyName
If you specify this parameter, the value of this property is returned. If you do not specify this parameter, all JVM properties will return in list format. Each element in the list is a property name and value pair. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showJVMProperties {-serverName server1 -nodeName node1  -propertyName
 test.property}
```

- Using Jython string:

```
AdminTask.showJVMProperties('[-serverName server1 -nodeName node1  -propertyName
 test.property]')
```

- Using Jython list:

```
AdminTask.showJVMProperties(['-serverName', 'server1', '-nodeName', 'node1',
 '-propertyName', 'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showJVMProperties {-interactive}
```

- Using Jython string:

```
AdminTask.showJVMProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showJVMProperties (['-interactive'])
```

showJVMSystemProperties

Use the **showJVMSystemProperties** command to show the Java virtual machine (JVM) system properties for the process of the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose JVM properties will be shown. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-propertyName
If you specify this parameter, the value of specified property is returned. If you do not specify this parameter, all properties will return in a list where each element is a property name and value pair. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showJVMSystemProperties {-serverName server1 -nodeName node1
 -propertyName test.property}
```

- Using Jython string:

```
AdminTask.showJVMSystemProperties('[-serverName server1 -nodeName node1
 -propertyName test.property]')
```

- Using Jython list:

```
AdminTask.showJVMSystemProperties(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName',  'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showJVMSystemProperties {-interactive}
```

- Using Jython string:

```
AdminTask.showJVMSystemProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showJVMSystemProperties (['-interactive'])
```

showProcessDefinition

Use the **showProcessDefinition** command to show the process definition of the server.

Target object

None

Required parameters

-serverName
Specifies the name of the server for which the process definition is shown. If only one server exists in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-propertyName
If you do not specify this parameter, all the process definitions of the server are returned in a list format where each element in the list is property name and value pair. If you specify this parameter, the property value of the property name that you specified is returned. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showProcessDefinition {-serverName server1 -nodeName node1 -propertyName
 test.property}
```

- Using Jython string:

```
AdminTask.showProcessDefinition('[-serverName server1 -nodeName node1 -propertyName
 test.property]')
```

- Using Jython list:

```
AdminTask.showProcessDefinition(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName',  'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showProcessDefinition {-interactive}
```

- Using Jython string:

```
AdminTask.showProcessDefinition ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showProcessDefinition (['-interactive'])
```

showServerInfo

The **showServerInfo** command returns the information for a server that you specify.

Target object

The configuration ID of the server. (required)

Parameters and return values

- Parameters: None
- Returns: A list of metadata.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showServerInfo server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)
```

- Using Jython string:

```
AdminTask.showServerInfo('server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)')
```

- Using Jython list:

```
AdminTask.showServerInfo('server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showServerInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showServerInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showServerInfo (['-interactive'])
```

showServerTypeInfo

The **showServerTypeInfo** command displays information about a specific server type.

Target object

Specifies a server type. For example: `APPLICATION_SERVER` (String, required)

Optional parameters

-version
Specifies the version of the templates that you want to list. For example, 6.0.0.0. (String, optional)

-serverType
Specifies if you want to list templates for a specific server type. (String, optional)

-name
Specifies whether to look for a specific template. (String, optional)

-queryExp
Specifies a key and value pair that you can use to find templates by properties. For example, `com.ibm.websphere.nodeOperatingSystem=os390`. (String[], optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showServerTypeInfo APPLICATION_SERVER
```

- Using Jython string:

```
AdminTask.showServerTypeInfo(APPLICATION_SERVER)
```

- Using Jython list:

```
AdminTask.showServerTypeInfo(APPLICATION_SERVER)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showServerTypeInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showServerTypeInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showServerTypeInfo (['-interactive'])
```

showTemplateInfo

Use the **showTemplateInfo** command to display the metadata information for a specific server template.

Target object

Specifies the configuration Id of the server type. (String, required)

Parameters and return values

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showTemplateInfo
 default(templates/servertypes/APPLICATION_SERVER/servers/default|server.xml)  {isSystemTemplate true} {name default}
 {com.ibm.websphere.baseProductVersion 6.0.0} {description {The WebSphere  Default Server Template}}
 {com.ibm.websphere.baseProductMinorVersion 0.0}
 {com.ibm.websphere.baseProductMajorVersion 6}  {com.ibm.websphere.nodeOperatingSystem {}} {isDefaultTemplate true}
```

- Using Jython string:

```
AdminTask.showTemplateInfo(default(templates/serverTypes/APPLICATION_SERVER/servers/default|server.xml))  '[[isSystemTemplate
 true] [com.ibm.websphere.baseProductVersion 6.0.0] [name default]  [com.ibm.websphere.baseProductMinorVersion 0.0] [description
 The WebSphere Default Server Template]
 [isDefaultTemplate true] [com.ibm.websphere.nodeOperatingSystem] [com.ibm.websphere.baseProductMajorVersion 6]]'
```

- Using Jython list:

```
AdminTask.showTemplateInfo(default(templates/serverTypes/APPLICATION_SERVER/servers/default|server.xml))  [['isSystemTemplate',
 'true'], ['com.ibm.websphere.baseProductVersion', '6.0.0'], ['name', 'default']  ['com.ibm.websphere.baseProductMinorVersion',
 '0.0'], ['description', 'The WebSphere
 Default Server Template']  ['isDefaultTemplate', 'true'], ['com.ibm.websphere.nodeOperatingSystem'],
 ['com.ibm.websphere.baseProductMajorVersion',  '6']]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showTemplateInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showTemplateInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showTemplateInfo (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

UnmanagedNodeCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the UnmanagedNodeCommands group can be used to create and query for managed and unmanaged nodes. An unmanaged node is a node that does not have a node agent or a deployment manager.

The UnmanagedNodeCommands command group for the AdminTask object includes the following commands:

- “createUnmanagedNode” on page 524
- “listManagedNodes” on page 524

createUnmanagedNode

Use the **create Unmanaged Node** command to create a new unmanaged node in the configuration. An unmanaged node is a node that does not have a node agent or a deployment manager. Unmanaged nodes can contain Web servers, such as IBM HTTP Server.

Target object

None

Parameters and return values

-nodeName
The name that will represent the node in the configuration repository. (String, required)

-hostName
The host name of the system associated with this node. (String, required)

-nodeOperatingSystem
The operating system in use on the system associated with this node. Valid entries include the following: `os400`, `aix`, `hpux`, `linux`, `solaris`, `windows`, and `os390`.(String required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask createUnmanagedNode {-nodeName myNode -hostName myHost -nodeOperatingSystem linux}
  ```
- Using Jython string:
  ```
  AdminTask.createUnmanagedNode ('[-nodeName jjNode -hostName jjHost -nodeOperatingSystem linux]')
  ```
- Using Jython list:
  ```
  AdminTask.createUnmanagedNode (['-nodeName', 'jjNode', '-hostName', 'jjHost', '-nodeOperatingSystem', 'linux'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask createUnmanaged Node {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.createUnmanaged Node ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.createUnmanaged Node (['-interactive'])
  ```

listManagedNodes

Use the **listManaged Nodes** command to list the managed nodes, nodes that have a node agent defined, in a configuration.

Target object

None

Parameters and return values

- Parameters: None

- Returns: List

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask listManagedNodes
  ```
- Using Jython string:
  ```
  AdminTask.listManagedNodes()
  ```
- Using Jython list:
  ```
  AdminTask.listManagedNodes()
  ```

listUnmanagedNodes

Use the **list Unmanaged Nodes** command to list the unmanaged nodes in a configuration.

Target object

None

Parameters and return values
- Parameters: None
- Returns: List

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask listUnmanagedNodes
  ```
- Using Jython string:
  ```
  AdminTask.listUnmanagedNodes()
  ```
- Using Jython list:
  ```
  AdminTask.listUnmanagedNodes()
  ```

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask listUnmanagedNodes {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listUnmanagedNodes ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listUnmanagedNodes (['-interactive'])
  ```

removeUnmanagedNode

Use the **remove Unmanaged Node** command to remove an unmanaged node from the configuration.

Target object

None

Parameters and return values

-nodeName
The name of the unmanaged node. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeUnmanaged Node {-nodeName myNode }
```

- Using Jython string:

```
AdminTask.removeUnmanaged Node('[-nodeName myNode]')
```

- Using Jython list:

```
AdminTask.removeUnmanaged Node(['-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeUnmanaged Node {-interactive}
```

- Using Jython string:

```
AdminTask.createUnmanaged Node ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createUnmanaged Node (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

ConfigArchiveOperations command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure servers in your environment. The commands and parameters in the ConfigArchiveOperations group can be used to export or import server configurations and entire cell configurations.

The ConfigArchiveOperations command group for the AdminTask object includes the following commands:

- "exportProxyProfile"
- "exportProxyServer" on page 527
- "exportServer" on page 528
- "exportWasprofile" on page 529
- "importProxyProfile" on page 529
- "importProxyServer" on page 530
- "importServer" on page 531
- "importWasprofile" on page 532

exportProxyProfile

Use the exportProxyProfile command to export the entire cell configuration of a secure proxy server to a configuration archive. The exportProxyProfile command does not work between the distributed and z/OS platforms.

Target object

None.

Required parameters

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportProxyProfile('-archive /myCell.car')
```

- Using Jython list:

```
AdminTask.exportProxyProfile('-archive', '/myCell.car')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportProxyProfile('-interactive')
```

exportProxyServer

Use the exportProxyServer command to export the secure proxy server configuration to a node that is defined in the configuration archive. The command exports the metadata file of the node where the server resides. You can use this information later when you import the configuration archive to verify that the target node is compatible to the node from which you are exporting the server.

The exportProxyServer command virtualizes the server configuration and exports a server to a configuration archive. This process breaks any existing associations between the server configurations in the configuration archive and the configurations in the system.

Target object

None

Required parameters

-archive
Specifies the fully qualified path of the exported configuration archive. (String, required)

-serverName
Specifies the secure proxy server name. (String, required)

Optional parameters

-nodeName
Specifies the node name of the secure proxy server. This parameter is only required if the secure proxy server name is not unique across the cell. (String, optional)

Return value

The command does not return output.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.exportProxyServer('[-archive c:\myProxyServer.car -nodeName node1
 -serverName server1]')
```

- Using Jython list:

```
AdminTask.exportProxyServer(['-archive', 'c:\myProxyServer.car', '-nodeName',
 'node1', '-serverName', 'server1'])
```

- Using Jython string:

```
AdminTask.exportProxyServer('[-archive /myProxyServer.car -nodeName node1
 -serverName server1]')
```

- Using Jython list:

```
AdminTask.exportProxyServer(['-archive', '/myProxyServer.car', '-nodeName', 'node1',
 '-serverName', 'server1'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.exportServer ('-interactive')
```

exportServer

Use the exportServer command to export the server configuration to a node that is defined in the configuration archive. Use the exportProxyServer command to export a proxy server configuration.

The exportServer command virtualizes the server configuration and exports a server to a configuration archive. This process breaks any existing associations between the server configurations in the configuration archive and the configurations in the system. This process also removes applications from the server that you specify, breaks the relationship between the server that you specify and the core group of the server, cluster, or service integration bus member.

The exportServer command exports the metadata file of the node where the server resides. You can use this information later when you import the configuration archive to verify that the target node is compatible to the node from which you are exporting the server.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified path of the exported configuration archive. (String, required)

-nodeName
Specifies the node name of the server. This parameter is only required when the server name is not unique across the cell. (String, optional)

-serverName
Specifies the server name. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask exportServer {-archive c:\myServer.car -nodeName node1 -serverName
 server1}
```

- Using Jython string:

```
AdminTask.exportServer ('[-archive c:\myServer.car -nodeName node1 -serverName
 server1]')
```

- Using Jython list:

```
AdminTask.exportServer (['-archive', 'c:\ myServer.car', '-nodeName', 'node1',
 '-serverName', 'server1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exportServer {-interactive}
```

- Using Jython string:

```
AdminTask.exportServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.exportServer (['-interactive'])
```

exportWasprofile

Use the exportWasprofile command to export the entire cell configuration to a configuration archive. The exportWasprofile command does not work between the distributed and z/OS platforms. Only a base server configuration with single node is supported for this command. Use the exportProxyProfile command to export a secure proxy server configuration.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask exportWasprofile {-archive c:\myCell.car}
```

- Using Jython string:

```
AdminTask.exportWasprofile('[-archive c:\myCell.car]')
```

- Using Jython list:

```
AdminTask.exportWasprofile(['-archive', 'c:\myCell.car'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exportWasprofile {-interactive}
```

- Using Jython string:

```
AdminTask.exportWasprofile ('[-interactive]')
```

- Using Jython list:

```
AdminTask.exportWasprofile (['-interactive'])
```

importProxyProfile

Use the importProxyProfile command to import a cell configuration in the configuration archive to the system. Only a base single server configuration is supported for this command. The importProxyProfile command does not work between the distributed and z/OS platforms.

Target object

None.

Required parameters

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Optional parameters

-deleteExistingServers
Specifies whether to replace the existing secure proxy servers in the profile with the servers in the imported proxy profile. Specify `true` to overwrite the existing servers. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.importProxyProfile('-archive /myCell.car -deleteExistingServers
 true')
```

- Using Jython list:

```
AdminTask.importProxyProfile('-archive', '/myCell.car', '-deleteExistingServers', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importProxyProfile('-interactive')
```

importProxyServer

Use the importProxyServer command to import a secure proxy server that resides in a configuration archive to the system. This command imports all the server scope configurations defined in the configuration archive to system configuration.

Target object

None

Required parameters

-archive
Specifies the fully qualified path of the configuration archive to import. (String, required)

Optional parameters

-nodeInArchive
Specifies the node name of the server defined in the configuration archive. Specify a value for this parameter if multiple nodes exist in the configuration archive. (String, optional)

-serverInArchive
Specifies the name of the secure proxy server defined in the configuration archive. Specify a value for this parameter if multiple secure proxy servers exist in the archive. (String, optional)

-deleteExistingServer
Specifies whether to delete and replace an existing server if it has the same name as the server to import. Set the value of this command to `true` to overwrite existing servers with the same name. (String, optional)

-nodeName
Specifies the name of the node to which the secure proxy server is imported. This parameter is only required if the secure proxy server name is not unique across the cell. (String, optional)

-serverName
Specifies the secure proxy server name. If the server name that you specify matches an existing server name under the node, an exception is created. (String, optional)

-coreGroup
Specifies the core group name to which the secure proxy server belongs. (String, optional)

Return value

The command does not return output.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.importProxyServer('[-archive c:\myProxyServer.car -nodeName node1
 -serverInArchive server1 -deleteExistingServer true]')
```

- Using Jython list:

```
AdminTask.importProxyServer(['-archive', 'c:\myProxyServer.car', '-nodeName',
 'node1', '-serverInArchive', 'server1', '-deleteExistingServer', 'true'])
```

- Using Jython string:

```
AdminTask.importProxyServer('[-archive /myProxyServer.car -nodeName node1
 -serverInArchive server1 -deleteExistingServer true]')
```

- Using Jython list:

```
AdminTask.importProxyServer(['-archive', '/myProxyServer.car', '-nodeName', 'node1',
 '-serverInArchive', 'server1', 'server1', '-deleteExistingServer', 'true'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.importProxyServer('-interactive')
```

importServer

Use the importServer command to import a server that resides in a configuration archive to the system. This command imports all the server scope configurations defined in the configuration archive to system configuration. Use the importProxyServer command to import a secure proxy server configuration.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified path of the configuration archive. (String, required)

-nodeInArchive
Specifies the node name of the server defined in the configuration archive. (String, optional if there is only one node defined in the configuration archive, required if there are multiple nodes defined in the configuration archive)

-serverInArchive
Specifies the name of the server defined in the configuration archive. (String, optional if there is only one server defined on the specified *nodeInConfiguration* archive, required if there are multiple servers defined under the specified *nodeInConfiguration* archive)

-nodeName
Specifies the node name where the server is imported. (String, optional if there is only one node)

-serverName
Specifies the server name where the server is imported. If the server name that you specify matches an existing server name under the node, an exception is created. (String, optional, default:serverInArchive)

-coreGroup
Specifies the core group name to which the server should belong. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask importServer {-archive c:\myServer.car -nodeInArchive node1
 -serverInArchive server1}
```

- Using Jython string:

```
AdminTask.importServer ('[-archive c:\myServer.car -nodeInArchive node1
 -serverInArchive server1]')
```

- Using Jython list:

```
AdminTask.importServer (['-archive', 'c:\myServer.car', '-nodeInArchive', 'node1',
 '-serverInArchive', 'server1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask importServer {-interactive}
```

- Using Jython string:

```
AdminTask.importServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.importServer (['-interactive'])
```

importWasprofile

Use the importWasprofile command to import a cell configuration in the configuration archive to the system. Only a base server configuration with single node is supported for this command. Use the importProxyProfile command to import a secure proxy server profile.

The importWasprofile command does not work between the distributed and z/OS platforms.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified file path of the configuration archive. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 9. Clustering servers with scripting

You can use scripting and the wsadmin tool to cluster application servers, generic servers, Web servers, and proxy servers.

About this task

This topic contains the following tasks:

Creating clusters using scripting

Use the wsadmin tool to create application server, generic server, and proxy server clusters. A cluster is a set of servers that you manage together as a way to balance workload.

Before you begin

There are multiple ways to complete this task. This topic uses the AdminConfig object to create clusters in your environment. Alternatively, you can use the ClusterConfigCommands command group for the AdminTask object or the createCluster script in the AdminClusterManagement script library to create and configure clusters.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the server to convert to a cluster.

   Determine the configuration ID of the server of interest and assign it to the server variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     ```

3. Convert the existing server to a cluster.

   Use the **convertToCluster** command for the AdminConfig object to convert the existing server to a cluster. Specify the name of the server and the name to assign to the cluster, as the following example demonstrates.

   The resourcesScope parameter is not a required parameter. If it is not included in the command the server resources are copied to the cluster scope, with the cluster resources being overwritten. The following values can be specified for this parameter:

   - both, null, or empty String: The server resources are copied to the cluster scope. This results in the cluster scope resources being over written by the server scope resources. The resources on the server scope remain unchanged.
   - cluster: The server resources are moved to the cluster scope. This results in the cluster scope resources being over written by the server scope resources. The resources on the server scope are deleted with the exception with the EJBTimer attributes in the resources.xml and the resources-cei.xml resources remaining on the server scope.

- server: Any resources from both the cluster scope and the server scope remain where they are unchanged.
- Using Jacl:

  ```
  $AdminConfig convertToCluster $server myCluster1 resourcesScope
  ```

- Using Jython:

  ```
  AdminConfig.convertToCluster(server, 'myCluster1' resourcesScope)
  ```

  Example output:

  ```
  myCluster1(cells/mycell/cluster/myCluster1|cluster.xml#ClusterMember_1)
  ```

4. Save the configuration changes.
5. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, as the following example demonstrates:

   ```
   AdminNodeManagement.syncActiveNodes()
   ```

Modifying cluster member templates using scripting

Use the AdminConfig object and the wsadmin tool to modify cluster member templates for application server, generic server, and proxy server clusters.

About this task

A copy of the first cluster member that you create is stored in the cluster scope as a template. You can create the first cluster member using any existing server as a template or a default server template. You can also create a first cluster member when you create the cluster by converting a server to a cluster. When you create a first cluster member, the template of the cluster member is stored under the scope of the cluster. Additional cluster members are created using the cluster member template stored in the cluster scope.

A cluster can be either homogeneous or heterogeneous in nature. A homogeneous cluster spans nodes that are of the same product version. A heterogeneous cluster spans nodes of different products versions. Since a cluster can contain members from nodes that run on different versions of the product, one template will be stored for each version of the application server node that is configured as a cluster member. The cluster member template will not exist for a given node version until you create a first member in a node of the same version. For example, if a cluster contains several V7 nodes and several V6.1 nodes, there will be one cluster member template for the V7 nodes and one for the V6.1 nodes, such as the following:

- The $*WAS_HOME*/config/clusters/clusterName/servers/V7.0MemberTemplate template will be used as the template for any member that is created in a V7.0 node.
- The $*WAS_HOME*/config/clusters/clusterName/servers/V6.1MemberTemplate template will be used as the template for any member that is created in a V6.1 node.
- The $*WAS_HOME*/config/clusters/clusterName/servers/V6MemberTemplate template will be used as the template for any member that is created in a V6 node.
- The $*WAS_HOME*/config/clusters/clusterName/servers/V5MemberTemplate template will be used as the template for any member that is created in a V5 node.

Therefore, when you make a configuration change to cluster members, you must make the same configuration change to the template that is stored in the corresponding cluster scope in order to keep the template in sync with the existing members. Similarly, when you make a configuration change to the template, you should make the same configuration change to existing cluster members.

You can modify a cluster member template using the wsadmin tool similar to how you modify a server. You cannot modify a cluster member template using the administrative console. Perform the following steps to modify a cluster member template using the wsadmin tool:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Obtain the cluster template under the cluster scope. For example, the following example obtains the version 6.1 cluster member template for the cluster1 cluster:
   - Using Jacl:

     ```
     set c [$AdminConfig listTemplates Server cluster1/servers/V7.0]

     puts [$AdminConfig showall $c]
     ```

     Using Jython:

     ```
     c = AdminConfig.listTemplates('Server','cluster1/servers/V7.0')

     print AdminConfig.showall(c)
     ```
3. Modify the attributes of the template. For example:
   - Using Jacl:

     ```
     $AdminConfig modify $c {{attrName attrVal}}
     ```

     Using Jython:

     ```
     AdminConfig.modify(c, [[attrName, attrVal]])
     ```
4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Creating cluster members using scripting

Use the wsadmin tool to create cluster members in application server, generic server, Web server, and proxy server clusters.

Before you begin

There are multiple ways to complete this task. This topic uses the AdminConfig object to create cluster members in your environment. Alternatively, you can use the ClusterConfigCommands command group for the AdminTask object or the createClusterMember script in the AdminClusterManagement script library to create and configure clusters.

About this task

The template options are available only for the first cluster member that you create. All cluster members that you create after the first member will be identical. A template is stored in the cluster scope that you must use to create additional cluster members.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask createClusterMember {-interactive} resourcesScope
       ```
     - Using Jython:

       ```
       AdminTask.createClusterMember ('[-interactive]') resourcesScope
       ```

       The resourcesScope parameter is not a required parameter. If it is not included in the command the server resources are copied to the cluster scope, with the cluster resources being overwritten. The following values can be specified for this parameter:
       - both, null, or empty String: The server resources are copied to the cluster scope. This results in the cluster scope resources being over written by the server scope resources. The resources on the server scope remain unchanged.

- cluster: The server resources are moved to the cluster scope. This results in the cluster scope resources being over written by the server scope resources. The resources on the server scope are deleted with the exception with the EJBTimer attributes in the resources.xml and the resources-cei.xml resources remaining on the server scope.
- server: Any resources from both the cluster scope and the server scope remain where they are unchanged.

- Using the AdminConfig object:
  a. Identify the existing cluster and assign it to the cluster variable:
     – Using Jacl:
     ```
     set cluster [$AdminConfig getid /ServerCluster:myCluster1/]
     ```
     – Using Jython:
     ```
     cluster = AdminConfig.getid('/ServerCluster:myCluster1/')
     print cluster
     ```
     Example output:
     ```
     myCluster1(cells/mycell/cluster/myCluster1|cluster.xml#ServerCluster_1)
     ```
  b. Identify the node to create the new server and assign it to the node variable:
     – Using Jacl:
     ```
     set node [$AdminConfig getid /Node:mynode/]
     ```
     – Using Jython:
     ```
     node = AdminConfig.getid('/Node:mynode/')
     print node
     ```
     Example output:
     ```
     mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
     ```
  c. (Optional) Identify the cluster member template and assign it to the serverTemplate variable:
     – Using Jacl:
     ```
     set serverTemplate [$AdminConfig listTemplates Server]
     ```
     – Using Jython:
     ```
     serverTemplate = AdminConfig.listTemplates('Server')
     print serverTemplate
     ```
     Example output:
     ```
     server1(templates/default/nodes/servers/server1|server.xml#Server_1)
     ```
  d. Create the new cluster member, by using the **createClusterMember** command.
     – The following example creates the new cluster member, passing in the existing cluster configuration ID, existing node configuration ID, and the new member attributes:
       - Using Jacl:
       ```
       $AdminConfig createClusterMember $cluster $node {{memberName clusterMember1}}
       ```
       - Using Jython:
       ```
       AdminConfig.createClusterMember(cluster, node, [['memberName', 'clusterMember1']])
       ```
     – The following example creates the new cluster member with a template, passing in the existing cluster configuration ID, existing node configuration ID, the new member attributes, and the template ID:
       - Using Jacl:
       ```
       $AdminConfig createClusterMember $cluster $node
       {{memberName clusterMember1}} $serverTemplate
       ```
       - Using Jython:
       ```
       print AdminConfig.createClusterMember(cluster, node,
       [['memberName',  'clusterMember1']], serverTemplate)
       ```
       Example output:
       ```
       clusterMember1(cells/mycell/clusters/myCluster1|cluster.xml$ClusterMember_2)
       ```

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, as the following example demonstrates:

   ```
   AdminNodeManagement.syncActiveNodes()
   ```

Creating clusters without cluster members using scripting

You can use the wsadmin tool to create application server, generic server, Web server, and proxy server clusters without cluster members.

Before you begin

There are multiple ways to complete this task. This topic uses the AdminConfig or AdminTask objects to create clusters without cluster members in your environment. Alternatively, you can use the createClusterWithoutMember script in the AdminClusterManagement script library to create and configure clusters.

About this task

Perform the following steps to create a cluster without a cluster member:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask createCluster {-interactive}
       ```

     - Using Jython:

       ```
       AdminTask.createCluster (['-interactive'])
       ```

   - Using the AdminConfig object:
     a. Identify the cell configuration ID and set it to the s1 variable:
        - Using Jacl:

          ```
          set s1 [$AdminConfig getid /Cell:mycell/]
          ```

        - Using Jython:

          ```
          s1 = AdminConfig.getid('/Cell:mycell/')
          ```

     b. Create a new cluster without a cluster member:
        - Using Jacl:

          ```
          $AdminConfig create ServerCluster $s1 {{name ClusterName}}
          ```

        - Using Jython:

          ```
          print AdminConfig.create('ServerCluster', s1, '[[name ClusterName]]')
          ```

3. Save the configuration changes.
4. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, as the following example demonstrates:

   ```
   AdminNodeManagement.syncActiveNodes()
   ```

Starting clusters using scripting

Use the wsadmin tool to start application server, generic server, and proxy server clusters in the application server runtime.

Before you begin

There are multiple ways to complete this task. This topic uses the AdminControl object to start clusters in your environment. Alternatively, you can use the ClusterConfigCommands command group for the AdminTask object or the startSingleCluster, stopSingleCluster, rippleStartAllClusters, and rippleStartSingleCluster scripts in the AdminClusterManagement script library to administer clusters.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the ClusterMgr MBean and assign it to the clusterMgr variable.
   - Using Jacl:

     ```
     set clusterMgr [$AdminControl completeObjectName cell=mycell,type=ClusterMgr,*]
     ```
   - Using Jython:

     ```
     clusterMgr = AdminControl.completeObjectName('cell=mycell,type=ClusterMgr,*')
     print clusterMgr
     ```

   This command returns the ClusterMgr MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=ClusterMgr,mbeanIdentifier=ClusterMgr,
   type=ClusterMgr,process=dmgr
   ```
3. Refresh the list of clusters.
   - Using Jacl:

     ```
     $AdminControl invoke $clusterMgr retrieveClusters
     ```
   - Using Jython:

     ```
     AdminControl.invoke(clusterMgr, 'retrieveClusters')
     ```

   This command calls the retrieveClusters operation on the ClusterMgr MBean.
4. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:

     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```
   - Using Jython:

     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

   This command returns the Cluster MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
   ```
5. Start or RippleStart the cluster.
   - To start a cluster, use the following example. These commands invoke the start operation on the cluster MBean:
     –

       Using Jacl:

       ```
       $AdminControl invoke $cluster start
       ```
     – Using Jython:

       ```
       AdminControl.invoke(cluster, 'start')
       ```
   - Use the following example to RippleStart a cluster. RippleStart combines stopping and starting operations. It first stops and then restarts each member of the cluster. For example, your cluster contains 3 cluster members named server_1, server_2 and server_3. When you click RippleStart, server_1 stops and restarts, then server_2 stops and restarts, and finally server_3 stops and restarts. Use the RippleStart option instead of manually stopping and then starting all of the application servers in the cluster. The following commands invoke the rippleStart operation on the cluster MBean:
     –

Using Jacl:

```
$AdminControl invoke $cluster rippleStart
```

– Using Jython:

```
AdminControl.invoke(cluster, 'rippleStart')
```

Querying cluster state using scripting

You can query cluster states using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to query cluster state:

1. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:

     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```

   - Using Jython:

     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

   This command returns the Cluster MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
   ```

2. Query the cluster state.
   - Using Jacl:

     ```
     $AdminControl getAttribute $cluster state
     ```

   - Using Jython:

     ```
     AdminControl.getAttribute(cluster, 'state')
     ```

   This command returns the value of the run-time state attribute.

Stopping clusters using scripting

Use scripting and the wsadmin tool to stop application server, generic server, and proxy server clusters.

Before you begin

There are multiple ways to complete this task. This topic uses the AdminControl object to stop clusters in your application server runtime. Alternatively, you can use the ClusterConfigCommands command group for the AdminTask object or the immediateStopAllRunningClusters, immediateStopSingleCluster, stopAllClusters, and stopSingleCluster scripts in the AdminClusterManagement script library to administer clusters.

1. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:

     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```

   - Using Jython:

     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

This command returns the Cluster MBean.

Example output:

```
WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
```

2. Stop the cluster.
   - Using Jacl:

     ```
     $AdminControl invoke $cluster stop
     ```

   - Using Jython:

     ```
     AdminControl.invoke(cluster, 'stop')
     ```

   This command invokes the stop operation on the Cluster MBean.

ClusterConfigCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to cluster application servers, generic servers, and proxy servers using scripting. The commands and parameters in the ClusterConfigCommands group can be used to create and delete server clusters and groups of servers known as cluster members.

The ClusterConfigCommands command group for the AdminTask object includes the following commands:

createCluster

The createCluster command creates a new server cluster. A server cluster consists of a group of servers that are referred to as *cluster members*. Optionally, a replication domain can be created for the new cluster, and an existing server can be included as the first cluster member. You can also use the createCluster command to apply proxy server settings to the cluster.

Target object

None

Steps

-clusterConfig (required)
Specifies the following configuration information for the new server cluster:

-clusterName
Specifies the name of the server cluster. (String)

-preferLocal
Optionally specifies whether to enable or disable node-scoped routing optimization within the cluster. The default value is `false`. Specify `true` to enable node-scoped routing optimization. (Boolean)

-clusterType
Optionally specifies the type of the server cluster to create. The default type is `APPLICATION_SERVER`. Valid values for this parameter include: `APPLICATION_SERVER`, `PROXY_SERVER`, and `ONDEMAND_ROUTER`. (String)

-replicationDomain (optional)
The system uses the replication domain properties for HTTP session data replication.

-createDomain
Specifies whether to create a replication domain in your cluster configuration. The default value is `false`. Specify `true` to create a replication domain in your cluster configuration. (Boolean)

-convertServer (optional)
Specifies information about an existing application server to convert to be the first member of the cluster. This command step is optional. The following parameters can be specified for this step:

-serverNode
The name of the node with the server to be converted to the first cluster member. You must also specify the serverName parameter. (String)

-serverName
The name of the application server to be converted to the first cluster member. You must also specify the serverNode parameter. (String)

-memberWeight
The weight of the cluster member. The weight controls the amount of work directed to the application server. If the weight is greater than the weight assigned to other cluster members, the server will receive a larger share of the workload. The value is a number between 0 and 100. If none is specified, the default is `2`. (Integer)

-nodeGroup
The name of the node group which this cluster member″s node, and all future cluster members″ nodes, must belong to. All cluster members must reside on nodes in the same node group. If specified, it must be one of the node groups which this member″s node belongs to. If not specified, the default value will be the first node group listed for this member's node. (String)

-replicatorEntry
Specifies whether to enable HTTP session data replication. The default value is `false`. Specify `true` to enable HTTP session data replication. You must specify this parameter if the createDomain parameter was set to `true` in the replicationDomain command step. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createCluster {-clusterConfig {{-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER}}}
$AdminTask createCluster {-clusterConfig {{-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER}} -replicationDomain {{-createDomain true}}}
$AdminTask createCluster {-clusterConfig {{-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER}} -convertServer {{-serverNode node1 -serverName server1}}}
```

- Using Jython string:

```
AdminTask.createCluster('[-clusterConfig [[-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER]]]')
AdminTask.createCluster('[-clusterConfig [[-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER]] -replicationDomain [[true]]]')
AdminTask.createCluster('[-clusterConfig [[-clusterName cluster1 -preferLocal true
 -clusterType APPLICATION_SERVER]] -convertServer [[-serverNode node1 -serverName server1]]]')
```

- Using Jython list:

```
AdminTask.createCluster(['-clusterConfig', '[[-clusterName cluster1 -preferLocal
 true -clusterType APPLICATION_SERVER]]'])
AdminTask.createCluster(['-clusterConfig', '[[-clusterName cluster1 -preferLocal true
  -clusterType APPLICATION_SERVER]]', '-replicationDomain', '[[-createDomain true]]'])
AdminTask.createCluster(['-clusterConfig', '[[-clusterName cluster1 -preferLocal
 true -clusterType APPLICATION_SERVER]]', '-convertServer', '[[-serverNode node1 -serverName server1]]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCluster {-interactive}
```

- Using Jython:

```
AdminTask.createCluster ('-interactive')
```

createClusterMember

The createClusterMember command creates a member of a server cluster. A cluster member is an application server that belongs to a cluster. If this is the first member of the cluster, you must specify a template to use as the model for the cluster member. The template can be either a default server template, or an existing application server.

The first cluster member is used as a template to create subsequent members in the cluster. When you create a first cluster member, the template of the cluster member is stored under the scope of the cluster.

Since a cluster can contain members from nodes that run different versions of the application server, the following conditions apply:

- The system stores one template for each version of the node that is already configured as a cluster member.
- The cluster member template will not exist for a given version of node until a first member is created in any node of the same version. For example, if a cluster contains some V6.1 nodes and some V6.0.x nodes, there will be one cluster member template for the V6.1 node and one cluster member template for the V6.0.x node.
- The following template will be used for members that are created for a V6.1 node: $*WAS_HOME*/config/templates/clusters/clusterName/servers/V6.1MemberTemplate.
- The following template will be used for members that are created for a V6.0.x node: $*WAS_HOME*/config/templates/clusters/clusterName/servers/V6MemberTemplate.
- The following template will be used for members that are created for a V5.x node: $*WAS_HOME*/config/clusters/clusterName/servers/V5MemberTemplate.
- When you make a configuration change to members in a cluster, you must make the same configuration change to the template that is stored in the cluster scope that corresponds.

Target object

Optionally specifies the configuration ID of the cluster to which the new member belongs. If you do not specify the configuration ID, you must specify the clusterName parameter. Use the getid command for the AdminConfig object to get the configuration ID of the cluster of interest.

Required parameters

-clusterName
The name of the cluster to which the new member will belong. If you do not specify this parameter, you must specify the cluster object ID in the command target. (String)

Steps

-memberConfig (required)
Specifies the configuration of a new member of the cluster.

memberNode
Specifies the node on which the system creates the cluster member. (String)

memberName
Specifies the name of the new cluster member. (String)

memberWeight
Optionally specifies the starting weight of the cluster member. (Integer)

memberUUID
Optionally specifies the UUID of the cluster member. (String)

genUniquePorts
Optionally specifies whether the system generates unique port numbers for each HTTP transport defined in the server. The new server will not have HTTP transports which conflict with any other servers defined on the same node. The default value is `true`. If you do not want to generate unique port numbers, specify the value as `false`. (Boolean)

replicatorEntry
Optionally specifies whether the system creates a replicator entry for the new cluster member in the cluster replication domain. A replicator entry is used to provide HTTP session data replication. This command parameter is optional. The value is true or false which indicates whether the entry will be created. The default value is `false`. You can specify this parameter only if a replication domain has been created for the cluster. (Boolean)

-firstMember (optional)
Specifies additional information required to configure the first member of the cluster.

templateName
Optionally specifies The name of an application server template to use when creating the new cluster member. If you specify a template, you cannot specify the templateServerNode and templateServerName parameters to use an existing application server as a template. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters in this step. (String)

templateServerNode
Optionally specifies the name of the node with an existing application server to use as the template when creating the new cluster member. If you specify the templateServerNode parameter, you must also specify the templateServerName parameter, and you cannot specify the templateName parameter. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters, in this step. (String)

templateServerName
Optionally specifies the name of the existing application server to use as the model when creating the new cluster member. If you specify the templateServerName parameter, you must also specify the templateServerNode parameter, and you cannot specify the templateName parameter. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters, in this command step. (String)

nodeGroup
Optionally specifies the name of the node group to which the new cluster member and each additional cluster member belongs. Each cluster member must reside on nodes in the same node group. If specified, it must be one of the node groups which this member node belongs to. If you do not specify this parameter, the system assigns the first node group listed for the member node. (String)

coreGroup
Optionally specifies the name of the core group to which the new cluster member and each additional cluster member belongs. Each cluster members must belong to the same core group. If you do not specify this parameter, the system assigns the default core group in the cell. (String)

Examples

Batch mode example usage:

- Using Jacl:

  First member creation using template name:

```
$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{-memberNode node1 -memberName member1
  -genUniquePorts true -replicatorEntry false}} -firstmember {{-templateName serverTemplateName}}}
```

First member creation using server and node for template:

```
$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{-memberNode node1 -memberName member1
 -genUniquePorts true -replicatorEntry false}} -firstmember
 {{-templateServerNode node1 -templateServerName server1}}}
```

Second member creation:

```
$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{-memberNode node1 -memberName member2
  -genUniquePorts true -replicatorEntry false}}}
```

- Using Jython string:

  First member creation using template name:

```
AdminTask.createClusterMember('[-clusterName cluster1 -memberConfig [[-memberNode node1 -memberName member1
  -genUniquePorts true -replicatorEntry false]] -firstMember [[-templateName
 serverTemplateName]]]')
```

First member creation using server and node for template:

```
AdminTask.createClusterMember('[-clusterName cluster1 -memberConfig [[-memberNode node1 -memberName member1
  -genUniquePorts true -replicatorEntry false]] -firstMember
 [[-templateServerNode node1 -templateServerName server1]]]')
```

Second member creation:

```
AdminTask.createClusterMember('[-clusterName cluster1 -memberConfig [[-memberNode node1 -memberName member1
  -genUniquePorts true -replicatorEntry false]]]')
```

- Using Jython list:

  First member creation using template name:

```
AdminTask.createClusterMember(['-clusterName', 'cluster1',  '-memberConfig',
 '[[-memberNode node1 -memberName member1 -genUniquePorts true -replicatorEntry false]]',
 '-firstMember', '[[-templateName serverTemplateName]]'])
```

  First member creation using server and node for template:

```
AdminTask.createClusterMember(['-clusterName', 'cluster1', '-memberConfig', '[[-memberNode
 node1 -memberName member1 -genUniquePorts true -replicatorEntry false]]', '-firstMember',
 '[[-templateServerNode node1 -templateServerName server1]]'])
```

  Second member creation:

```
AdminTask.createClusterMember(['-clusterName', 'cluster1', '-memberConfig', '[[-memberNode
 node1 -memberName member1 -genUniquePorts true -replicatorEntry false]]'])
```

  Interactive mode example usage:

  – Using Jacl:

```
$AdminTask createClusterMember {-interactive}
```

  – Using Jython:

```
AdminTask.createClusterMember ('-interactive')
```

deleteCluster

The deleteCluster command deletes the configuration of a server cluster. A server cluster consists of a group of servers that are referred to as *cluster members*. The system deletes each cluster member for the cluster of interest.

Use the deleteClusterMember command to delete the configuration of an individual cluster member.

Target object

Optionally specifies the configuration object ID of the cluster to delete. If you do not specify the object ID for the cluster, then you must specify the clusterName parameter. Use the getid command for the AdminConfig object to get the configuration ID of the cluster.

Required parameters

-clusterName
Specifies the name of the cluster to delete. If you specify the configuration ID of the cluster, do not specify a value for the clusterName parameter. (String)

Steps

-replicationDomain (optional step)

-deleteRepDomain
Specifies whether to delete the replication domain. The default value is `false`. Specify `true` to delete the replication domain. (Boolean)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteCluster {-clusterName cluster1 }
$AdminTask deleteCluster {-clusterName cluster1 -replicationDomain {{-deleteRepDomain true}}}
```

- Using Jython string:

```
AdminTask.deleteCluster('[-clusterName cluster1]')
AdminTask.deleteCluster('[-clusterName cluster1 -replicationDomain [[-deleteRepDomain
 true]]]')
```

- Using Jython list:

```
AdminTask.deleteCluster(['-clusterName', 'cluster1'])
AdminTask.deleteCluster(['-clusterName', 'cluster1', '-replicationDomain',
 '[[-deleteRepDomain true]]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCluster -interactive
```

- Using Jython:

```
AdminTask.deleteCluster ('-interactive')
```

deleteClusterMember

The deleteClusterMember command deletes the configuration of a cluster member. A cluster member is an application server that belongs to a server cluster.

Use the deleteCluster command to delete the configuration of a cluster.

Target object

Optionally specifies the configuration object ID of the cluster member to delete. If you do not specify the configuration ID, then you must specify the clusterName, memberNode and memberName parameters. Use the getid command for the AdminConfig object to get the configuration ID of the cluster.

Required parameters

-clusterName
Specifies the name of the cluster to which the member of interest belongs. If this parameter is specified, then the memberName and memberNode parameters must also be specified. If this is not specified, then the member object ID must be specified in the command target. (String)

-memberNode
Specifies the name of the node to which the cluster member belongs. If this parameter is specified, then the memberName and clusterName parameters must also be specified. If this is not specified, then the cluster member object ID must be specified in the command target. (String)

-memberName
Specifies the server name of the member to delete from the cluster. If this parameter is specified, then the clusterName and memberNode parameters must also be specified. If this is not specified, then the member object ID must be specified in the command target. (String)

Steps

replicatorEntry (optional)

Specifies the removal of a replicator entry for this cluster member. This command step is optional. The following parameters can be specified for this step:

-deleteEntry

Delete the replicator entry having the cluster member name from the cluster replication domain. Specify the value as `true` to delete the replicator entry. The default value is `false`.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteClusterMember {-clusterName cluster1 -memberNode node1 -memberName
 member1}

$AdminTask deleteClusterMember {-clusterName cluster1 -memberNode node1
 -memberName member2 -replicatorEntry {{-deleteEntry true}}}
```

- Using Jython string:

```
AdminTask.deleteClusterMember('[-clusterName cluster1 -memberNode node1 -memberName
 member1]')

AdminTask.deleteClusterMember('[-clusterName cluster1 -memberNode node1 -memberName
 member2 -replicatorEntry [[-deleteEntry true]]]')
```

- Using Jython list:

```
AdminTask.deleteClusterMember(['-clusterName', 'cluster1', '-memberNode', 'node1', '-memberName',
 'member1'])

AdminTask.deleteClusterMember(['-clusterName', 'cluster1', '-memberNode', 'node1',
 '-memberName', 'member2', '-replicatorEntry', '[[-deleteEntry true]]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteClusterMember -interactive
```

- Using Jython:

```
AdminTask.deleteClusterMember ('-interactive')
```

Related tasks

“Automating application configurations using the scripting library” on page 126
The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage applications in your environment.

“Creating cluster members using scripting” on page 537
Use the wsadmin tool to create cluster members in application server, generic server, Web server, and proxy server clusters.

Creating a proxy server cluster using the wsadmin command
This topic describes how to create a cluster of proxy servers, using the **wsadmin** command, that can route requests to applications in a cell.

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Cluster configuration scripts” on page 119
The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to configure clusters with or without cluster members, using a template, and to remove clusters from your configuration. You can run each script individually, or combine procedures to create custom automation scripts.

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 10. Configuring security with scripting

You can configure security with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. Read about “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

If you enable security for an application server cell, supply authentication information to communicate with servers. The `sas.client.props` and the `soap.client.props` files are located in the following properties directory for each application server profile:

- *profile_root*`/properties`
- The nature of the properties file updates required for running in secure mode depend on whether you connect with a Remote Method Invocation (RMI) connector, a JSR160RMI connector, an Inter-Process Communications (IPC) or a SOAP connector:
  - If you use a Remote Method Invocation (RMI) connector or a JSR160RMI connector, set the following properties in the `sas.client.props` file with the appropriate values:

    ```
    com.ibm.CORBA.loginUserid=
    com.ibm.CORBA.loginPassword=
    ```

    Also, set the following property:

    ```
    com.ibm.CORBA.loginSource=properties
    ```

    The default value for this property is `prompt` in the `sas.client.props` file. If you leave the default value, then a dialog box is displayed with a password prompt. If the script is running unattended, then the system stops.
  - If you use a SOAP connector, set the following properties in the `soap.client.props` file with the appropriate values:

    ```
    com.ibm.SOAP.securityEnabled=true
    com.ibm.SOAP.loginUserid=
    com.ibm.SOAP.loginPassword=
    ```
  - If you use an IPC connector, set the following properties in the `ipc.client.props` file with the appropriate values:

    ```
    com.ibm.IPC.loginUserid=
    com.ibm.IPC.loginPassword=
    ```
- Specify user and password information. Choose one of the following methods:
  - Specify user name and password on a command line, using the **-user** and **-password** commands, as the following examples demonstrate:

    ```
    wsadmin -conntype JSR160RMI -port 2809 -user u1 -password secret1
    ```
  - Specify user name and password in the `sas.client.props` file for an RMI connector, the `ipc.client.props` file for the IPC connector, or the `soap.client.props` file for a SOAP connector.

  If you specify user and password information on a command line and in the `sas.client.props` file or the `soap.client.props` file, the command line information overrides the information in the `props` file.

Enabling and disabling security using scripting

You can use scripting to enable or disable application security, global security, administrative security based on the LocalOS registry, and authentication mechanisms.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

The default profile sets up procedures so that you can enable and disable administrative security based on LocalOS registry.

- Use the isAppEnabled command to determine if application security is enabled or disabled, as the following example demonstrates:
  - Using Jacl:

    ```
    $AdminTask isAppSecurityEnabled {}
    ```
  - Using Jython:

    ```
    AdminTask.isAppSecurityEnabled()
    ```

  This command returns a value of `true` if appEnabled is set to `true`. Otherwise, returns a value of `false`.
- Use the isGlobalSecurityEnabled command to determine if administrative security is enabled or disabled, as the following example demonstrates:
  - Using Jacl:

    ```
    $AdminTask isGlobalSecurityEnabled{}
    ```
  - Using Jython:

    ```
    AdminTask.isGlobalSecurityEnabled()
    ```

  Returns a value of `true` if enabled is set to `true`. Otherwise, returns a value of `false`.
- Use the setGlobalSecurity command to set administrative security based on the passed in value, as the following example demonstrates:
  - Using Jacl:

    ```
    $AdminTask setGlobalSecurity {-enabled true}
    ```
  - Using Jython:

    ```
    AdminTask.setGlobalSecurity ('[-enabled true]')
    ```

  Returns a value of `true` if the enabled field in the WCCM security model is successfully updated. Otherwise, returns a value of `false`.
- Use the **help** command to find out the arguments that you need to provide with this call, as the following example demonstrates:
  - Using Jacl:

    ```
    securityon help
    ```

    Example output:

    ```
    Syntax: securityon user password
    ```
  - Using Jython:

    ```
    securityon()
    ```

    Example output:

    ```
    Syntax: securityon(user, password)
    ```
- Enable administrative security based on the LocalOS registry, as the following example demonstrates:
  - Using Jacl:

    ```
    securityon user1 password1
    ```
  - Using Jython:

    ```
    securityon('user1', 'password1')
    ```
- Disable administrative security based on the LocalOS registry, as the following example demonstrates:
  - Using Jacl:

```
securityoff
```

– Using Jython:

```
securityoff()
```

- Enable and disable LTPA and Kerberos authentication.

  Use the setActiveAuthMechanism command to set Kerberos as the authentication mechanism in the security configuration, as the following example demonstrates:

  ```
  AdminTask.setActiveAuthMechanism('-authMechanismType KRB5')
  ```

  Use the setActiveAuthMechanism command to set LTPA as the authentication mechanism in the security configuration, as the following example demonstrates:

  ```
  AdminTask.setActiveAuthMechanism('-authMechanismType LTPA')
  ```

  Additionally, there are sample scripts located in the <WAS_ROOT>/bin directory on how to enable and disable LTPA authentication. The scripts are:

  – LTPA_LDAPSecurityProcs.py (python script)

  – LTPA_LDAPSecurityProcs.jacl (jacl script)

  Note: The scripts hard code the type of LDAP server and base distinguished name (baseDN). The LDAP server type is hardcoded as IBM_DIRECTORY_SERVER and the baseDN is hardcoded as o=ibm,cn=us.

Enabling and disabling Java 2 security using scripting

You can enable or disable Java 2 security with scripting and the wsadmin tool.

About this task

There are two ways to enable or disable Java 2 security. You can use the commands for the AdminConfig object, or you can use the setAdminActiveSecuritySettings command for the AdminTask object.

1. Use the setAdminActiveSecuritySettings command for the AdminTask object to enable or disable Java 2 security.
   a. Launch the wsadmin scripting tool using the Jython scripting language.
   b. Use the getActiveSecuritySettings command to display the current security settings, including custom properties for global security, as the following example demonstrates:
      -
        – Using Jacl:

        ```
        $AdminTask getActiveSecuritySettings
        ```

        – Using Jython:

        ```
        AdminTask.getActiveSecuritySettings()
        ```

   c. Use the setActiveSecuritySettings command to enable or disable Java 2 security.

      The following examples enable Java 2 security:

      -
        – Using Jacl:

        ```
        $AdminTask setAdminActiveSecuritySettings {-enforceJava2Security true}
        ```

        – Using Jython:

        ```
        AdminTask.setAdminActiveSecuritySettings('-enforceJava2Security true')
        ```

      The following examples disable Java 2 security:

      -
        – Using Jacl:

        ```
        $AdminTask setAdminActiveSecuritySettings {-enforceJava2Security false}
        ```

        – Using Jython:

```
AdminTask.setAdminActiveSecuritySettings('-enforceJava2Security false')
```

d. Save the configuration changes.

e. Synchronize the node.

Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

- Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

```
AdminNodeManagement.syncNode("myNode")
```

2. Use the AdminConfig object to enable Java 2 security.

a. Launch the wsadmin scripting tool using the Jython scripting language.

b. Identify the security configuration object and assign it to the security variable, as the following example demonstrates:

- Using Jacl:

```
set security [$AdminConfig list Security]
```

- Using Jython:

```
security = AdminConfig.list('Security')
print security
```

Example output:

```
(cells/mycell|security.xml#Security_1)
```

c. Modify the enforceJava2Security attribute to enable or disable Java 2 security, as the following examples demonstrates:

- To enable Java 2 security:
  - Using Jacl:

```
$AdminConfig modify $security {{enforceJava2Security true}}
```

  - Using Jython:

```
AdminConfig.modify(security, [['enforceJava2Security', 'true']])
```

- To disable Java 2 security:
  - Using Jacl:

```
$AdminConfig modify $security {{enforceJava2Security false}}
```

  - Using Jython:

```
AdminConfig.modify(security, [['enforceJava2Security', 'false']])
```

d. Save the configuration changes.

e. Synchronize the node.

Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

- Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

```
AdminNodeManagement.syncNode("myNode")
```

Configuring multiple security domains using scripting

You can customize your security configuration at the cell, sever, or cluster level by configuring multiple security domains.

Before you begin

Users assigned to the administrator role can configure security domains. Verify that you have the appropriate administrative role before configuring security domains. Also, enable global security in your environment before configuring multiple security domains.

About this task

You can create multiple security domains to customize your security configuration. Use multiple security domains to achieve the following goals:

- Configure different security attributes for administrative and user applications within a cell
- Consolidate server configurations by managing different security configurations within a cell
- Restrict access between applications with different user registries, or configure trust relationships between applications to support communication across registries

1. Create a security domain. Create multiple security domains in your configuration. By creating multiple security domains, you can configure different security attributes for administrative and user applications within a cell environment.
2. Assign the security domain to one or a set of resources or scopes. Assign management resources to security domains. Set management resources to your security domains to customize your security configuration for a cell, server, or cluster.
3. Customize your security configuration by specifying attributes for your security domain. See the following examples of security attributes:
   - User registries to validate user credentials
   - Authorization for validating access to resources
   - Trust association interceptor (TAI) to authenticate a Web user using a reverse proxy server
   - Application and system JAAS login configurations
   - LTPA timeout settings
   - Application security enablement to provide application isolation and requirements for authenticating application users
   - Java 2 Security to increase overall system integrity by checking for permissions before allowing access to certain protected system resources
   - Remote Method Invocation over Internet Inter-ORB Protocol (RMI/IIOP) to invoke Web services through remote procedure calls
   - Custom properties

Configuring security domains using scripting

Use this topic to create multiple security domains in your configuration. By creating multiple security domains, you can configure different security attributes for administrative and user applications within a cell environment.

Before you begin

You must have the administrator role to configure security domains. Also, enable global security in your environment before configuring multiple security domains.

About this task

You can create multiple security domains to customize your security configuration. Use multiple security domains to achieve the following goals:

- Configure different security attributes for administrative and user applications within a cell
- Consolidate server configurations by managing different security configurations within a cell

- Restrict access between applications with different user registries, or configure trust relationships between applications to support communication across registries

Use the following steps to create a new security domain with the wsadmin tool:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Create a security domain.

   To create a security domain, you can create a new security domain, copy an existing security domain, or copy the existing global security configuration.

   - Use the createSecurityDomain command to create the domain-security.xml and domain-security-map.xml security files in the *profile_root*/config/waspolicies/default/*security_configuration_name* directory. No configuration data is added to the domain-security.xml file. Use the following Jython command example to create a security domain:

```
AdminTask.createSecurityDomain('-securityDomainName securityDomain1 -securityDomainDescription "handles user applications"')
```

     The command returns the object name of the security domain that was created, as the following example output demonstrates:

```
'waspolicies/default/securitydomains/mydomain:domain-security.xml#AppSecurity_1183132319126'
```

   - Use the copySecurityDomain command to create a new security domain with the attributes of an existing security domain. If the security configuration of the existing domain has an active user registry defined, then a new realm name for that registry must be used in the new security configuration. If a realm name is not specified with the copySecurityDomain command, then the command assigns a name. Specify the following parameters to copy an existing security domain:

Parameter	Description
-securityDomainName	Specifies the name of the new security domain to create (String, required)
-copyFromSecurityDomainName	Specifies the name of the existing security domain to copy (String, required)
-realmName	Specifies the name of the realm in the security domain to create. The system assigns the realm name to the active user registry in the security domain (String, optional)
-securityDomainDescription	Specifies a description for the security domain to create (String, optional)

     Use the following Jython command to copy an existing security domain:

```
AdminTask.copySecurityDomain('-securityDomainName copyOfDomain1 -copyFromSecurityDomainName securityDomain1')
```

     The command returns the object name of the new security domain, as the following example output demonstrates:

```
'waspolicies/default/securitydomains/copyOfDomain1:domain-security.xml#AppSecurity_1183132319186'
```

   - Use the copySecurityDomainFromGlobalSecurity command to create a security domain by copying the global security configuration. If the global security configuration has an active user registry defined, then a new realm name for that registry must be used for the security domain to create. If you do not specify a realm name, then the command assigns a name. Specify the following parameters to copy the global security configuration:

Parameter	Description
-securityDomainName	Specifies the name of the new security domain to create (String, required)
-realmName	Specifies the name of the realm in the security domain to create. The system assigns the realm name to the active user registry in the security domain (String, optional)
-securityDomainDescription	Specifies a description for the security domain to create (String, optional)

     Use the following Jython command to copy the global security configuration:

```
AdminTask.copySecurityDomainFromGlobalSecurity('-securityDomainName GScopy')
```

     The command returns the object name of the new security domain, as the following example output demonstrates:

```
'waspolicies/default/securitydomains/copyOfDomain1:domain-security.xml#AppSecurity_1183132319186'
```

3. Save your configuration changes.

What to do next

Use the wsadmin tool to map a scope to your security domain. Additionally, you can configure security artifacts in the newly created domain, by:

- configuring user registries.
- enabling application and Java EE security.
- setting Lightweight Third-Party Authentication (LTPA) timeout.
- configuring System and Application Java™ Authentication and Authorization Service (JAAS) login.
- configuring Java 2 Connector (J2C) authorization data.
- configuring Remote Method Invocation over Internet Inter-ORB Protocol (RMI/IIOP) security.

Configuring local operating system user registries using scripting

Use this topic to configure user registries for global security and security domain configurations using the wsadmin tool. You can define user registries at the global level and for multiple security domains.

Before you begin

You must meet the following requirements before configuring local operating system user registries:

- You must have the administrator or new admin role.
- Enable global security in your environment.
- To configure local operating system user registries for multiple security domains, you must configure at least one security domain.

About this task

Configure local operating system user registries to support use of the authentication mechanism with the user accounts database of the local operating system. You can specify local operating system user registries at the global level and at the security domain.

When you configure a user registry in the global security configuration, the administrator does not specify a realm name for the user registry. The system determines the realm name from the security runtime. The system typically specifies the hostname for local operating system registries.

In security domains, you can configure a different realm for a user registry configuration. For example, you can configure two registries that use the same LDAP server listening on the same port, but use different base distinguished names (baseDN). This allows the configuration to serve different sets of users and groups. To use this type of scenario, you must specify a realm name for each user registry configured for a domain. Because there can be multiple realms in your configuration, you can also specify a list of trusted realms. This allows communication between applications that use different realms.

Use the following steps to configure local operating system user registries for your global security configuration and for multiple security domains:

- Configure local operating system registries for global security configurations.
  1. Use the configureAdminLocalOSUserRegistry command and the following optional parameters to configure a local operating system user registry in your global security configuration:

Parameter	Description	Data type
-autoGenerateServerId	Specifies whether to automatically generate the server identity to use for internal process communication. To set a specific server identity, specify the -serverId parameter.	Boolean
-serverId	Specifies the user identity in the repository to use for internal process communication.	String

Parameter	Description	Data type
-serverIdPassword	Specifies the password that corresponds to the user identity.	String
-primaryAdminId	Specifies the name of the user with administrative privileges as defined in the registry. This parameter does not apply to security configurations. The user name must exist in the user registry repository.	String
-customProperties	Specifies a list of attribute and value pairs to store as custom properties on the user registry. Separate each attribute and value pair with a comma character (,), as the following syntax displays: `"attribute1=value1","attribute2=value2"`	String
-verifyRegistry	Specifies whether to verify the user registry. The default value is `true` and verification is automatically performed.	Boolean
-ignoreCase	Specifies whether to perform the case-sensitive authorization check. This only applies to the z/OS local operating system user registry.	Boolean

Use the following Jython example command to configure the local operating system registry for global security:

```
AdminTask.configureAdminLocalOSUserRegistry('-autoGenerateServerId true -primaryAdminId gsAdmin')
```

2. Configure the user registry to be the active user registry for the server.

   For example, the following Jython command sets the active user registry as the `LocalOSUserRegistry` registry for your global security configuration:

```
AdminTask.setAdminActiveSecuritySettings('-activeUserRegistry LocalOSUserRegistry')
```

3. Save your configuration changes.

- Configure local operating system registries for security domains.
  1. Determine the name of the security domain to configure.

     Use the listSecurityDomains command to list all security domains on the server, as the following Jython example demonstrates:

```
AdminTask.listSecurityDomains()
```

     If you want to configure the local operating system registry for a specific server, cluster, or cell, use the getSecurityDomainForResource command to display the security domain name for the management scope of interest. The following Jython example displays the name of the security domain configured at the cell-level:

```
AdminTask.getSecurityDomainForResource('-resourceName Cell=:Node=myNode:Server=myServer')
```

     For this example, the command returns the following output:

```
domain2
```

  2. Configure a local operating system user registry for a security domain.

     Use the configureAppLocalOSUserRegistry command and the following optional parameters to configure a local operating system user registry:

Parameter	Description	Data type
-securityDomainName	Specifies the unique name that identifies the security domain of interest.	String
-realmName	Specifies the name of the realm of the user registry.	String
-customProperties	Specifies a list of attribute and value pairs to store as custom properties on the user registry object. Separate each attribute and value pair with a comma character (,).	String
-verifyRegistry	Specifies whether to verify the user registry. The default value is `true`, and verification is automatically performed.	Boolean

Parameter	Description	Data type
-ignoreCase	Specifies whether to perform the case-sensitive authorization check. This only applies to the z/OS local operating system user registry.	Boolean

Use the following Jython command to configure the local operating system user registry for the `domain2` security domain:

```
AdminTask.configureAppLocalOSUserRegistry('-securityDomainName domain2 -realmName domain2Realm')
```

3. Configure the user registry to be the active user registry for the server.

   For example, the following Jython command sets the active user registry as the `LocalOSUserRegistry` registry for your security domain configuration:

```
AdminTask.setAppActiveSecuritySettings('-securityDomainName domain2 -activeUserRegistry LocalOSUserRegistry')
```

4. Save your configuration changes.

What to do next

Configuring custom user registries using scripting

Use this topic to configure custom user registries for global security and security domain configurations using the wsadmin tool. You can define custom user registries at the global level and for multiple security domains.

Before you begin

You must meet the following requirements before configuring custom user registries:

- You must have the administrator or new admin role.
- Enable global security in your environment.
- Implement and build the UserRegistry interface and configure a custom registry.
- To configure custom user registries for multiple security domains, you must configure at least one security domain.

About this task

WebSphere Application Server security supports stand-alone custom registries in addition to the local operating system registry, standalone Lightweight Directory Access Protocol (LDAP) registries, and federated repositories for authentication and authorization. A stand-alone custom-implemented registry uses the UserRegistry Java interface as provided by the product. A stand-alone custom registry can support any type of account repository from a relational database, flat file, and so on. You can specify custom user registries at the global level and at the security domain.

When you configure a user registry in the global security configuration, the administrator does not specify a realm name for the user registry. The system determines the realm name from the security run time. The realm name for custom registries is set by the custom registry.

Use the following Jython command to make the user registry the active user registry in the global security configuration:

```
AdminTask.setAdminActiveSecuritySettings('-activeUserRegistry CustomUserRegistry')
```

Use the following Jython command to make the user registry the active user registry in the application security configuration:

```
AdminTask.setAppActiveSecuritySettings('-securityDomainName domain2 -activeUserRegistry CustomUserRegistry')
```

In security domains, you can configure a different realm for a user registry configuration. For example, you can configure two registries that use the same LDAP server listening on the same port, but use different base distinguished names (baseDN). This method supports the configuration to serve different sets of

users and groups. To use this type of scenario, you must specify a realm name for each user registry configured for a domain. Multiple realms can exist in your configuration, and you can also specify a list of trusted realms. Communications between applications that use different realms is supported.

Use the following steps to configure custom user registries for your global security configuration and for multiple security domains:

- Configure custom user registries for global security configurations.

  Use the configureAdminCustomUserRegistry command and the following optional parameters to configure a custom user registry in your global security configuration:

Parameter	Description	Data Type
-autoGenerateServerId	Specifies whether to automatically generate the server identity to use for internal process communication. To set a specific server identity, specify the -serverId parameter.	Boolean
-serverId	Specifies the user identity in the repository to use for internal process communication.	String
-serverIdPassword	Specifies the password that corresponds to the user identity.	String
-primaryAdminId	Specifies the name of the user with administrative privileges as defined in the registry. This parameter does not apply to security configurations. The user name must exist in the user registry repository.	String
-customRegClass	Specifies the class name that implements the UserRegistry interface in the com.ibm.websphere.security class.	String
-ignoreCase	Specifies whether to require case sensitive authorization. Specify `true` to ignore case during authorization.	Boolean
-customProperties	Specifies a list of attribute and value pairs to store as custom properties on the user registry object. Separate each attribute and value pair with a comma character (`,`) as the following syntax displays: `"attribute1=value1","attribute2=value2"`	String
-verifyRegistry	Specifies whether to verify the user registry. The default value is `true` and verification is automatically performed.	Boolean

  Use the following Jython example command to configure the local operating system registry for global security:

```
AdminTask.configureAdminCustomUserRegistry('-autoGenerateServerId true -primaryAdminId gsAdmin')
```

- Configure custom user registries for security domains.
  1. Determine the name of the security domain to configure.

     Use the listSecurityDomains command to list all security domains on the server, as the following Jython example demonstrates:

```
AdminTask.listSecurityDomains()
```

  2. Configure a custom user registry for a security domain.

     Use the configureAppCustomUserRegistry command and the following optional parameters to configure a local custom user registry:

Parameter	Description	Data type
-securityDomainName	Specifies the unique name that identifies the security domain of interest.	String
-realmName	Specifies the name of the realm of the user registry.	String
-customRegClass	Specifies the class name that implements the UserRegistry interface in the com.ibm.websphere.security class.	String

Parameter	Description	Data type
-ignoreCase	Specifies whether to require case sensitive authorization. Specify `true` to ignore case during authorization.	Boolean
-customProperties	Specifies a list of attribute and value pairs to store as custom properties on the user registry object. Separate each attribute and value pair with a comma character (,) as the following syntax displays: `"attribute1=value1","attribute2=value2"`	String
-verifyRegistry	Specifies whether to verify the user registry. The default value is `true` and verification is automatically performed.	Boolean

Use the following Jython example command to configure the local operating system user registry for the `domain2` security domain:

```
AdminTask.configureAppCustomUserRegistry('-securityDomainName domain2 -realmName domain2Realm')
```

What to do next

Configuring JAAS login modules using scripting

Use this topic to use the wsadmin tool to configure and manage Java Authentication and Authorization Service (JAAS) login entries to allow communication between realms in a multiple security domain environment.

Before you begin

You must meet the following requirements before configuring local operating system user registries:

- You must have the administrator or new admin role.
- Enable global security in your environment.
- Configure multiple realms using security domains in your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure a JAAS login module.

   Use the configureJAASLoginEntry command to configure a Java Authentication and Authorization Service (JAAS) login entry in a security domain or in the global security configuration. You can use this command to modify existing JAAS login entries or to create new login entries.

   Specify the following parameters to configure the JAAS login module:

Parameter	Description
-loginEntryAlias	Specifies an alias that identifies the JAAS login entry in the configuration. (String, required)
-loginType	Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String, required)
-securityDomainName	Specifies the name of the security configuration. If you do not specify a security domain name, the system updates the global security configuration. (String, optional)
-loginModules	Specifies a comma (,) separated list of login module class names. Specify the list in the order that the system calls them. (String, optional)

Parameter	Description
-authStrategies	Optionally specifies the authentication behavior as authentication proceeds down the list of login modules. (String, optional) Specify one or many of the following values in a comma (,) separated list: • REQUIRED Specifies that the LoginModule module is required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list for each realm. • REQUISITE Specifies that the LoginModule module is required to succeed. If authentication is successful, the process continues down the LoginModule list in the realm entry. If authentication fails, control immediately returns to the application. Authentication does not proceed down the LoginModule list. • SUFFICIENT Specifies that the LoginModule module is not required to succeed. If authentication succeeds, control immediately returns to the application. Authentication does not proceed down the LoginModule list. If authentication fails, the process continues down the list. • OPTIONAL Specifies that the LoginModule module is not required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list.

Use the configureJAASLoginEntry command to configure the JAAS login module, as the following Jython example demonstrates:

```
AdminTask.configureJAASLoginEntry('[-securityDomainName testDomain
-loginType application -loginEntryAlias testLoginEntry -loginModules
"com.ibm.ws.security.common.auth.module.WSLoginModuleImpl" -authStrategies "REQUIRED"]')
```

3. Set custom properties for the JAAS login module.

   Use the configureLoginModule command to specify custom properties, modify the authentication strategy, or set the module to use a login module proxy. The following Jython command sets the `debug` and `delegate` custom properties for the `testLoginEntry` JAAS login entry:

```
AdminTask.configureLoginModule('[-securityDomainName testDomain -loginType application
-loginEntryAlias testLoginEntry -loginModule com.ibm.ws.security.common.auth.module.WSLoginModuleImpl
-customProperties ["debug=true","delegate=WSLogin"]]')
```

4. Save your configuration changes.

Configuring Common Secure Interoperability authentication using scripting

Use this topic to use the wsadmin tool to configure inbound and outbound communications using the Common Secure Interoperability protocol. Common Secure Interoperability Version 2 (CSIv2) supports increased vendor interoperability and additional features.

Before you begin

You must meet the following requirements before configuring local operating system user registries:

- You must have the administrator or new admin role.
- Enable global security in your environment.
- Configure multiple realms using security domains in your environment.
- Configure CSI inbound communication authentication.

  Inbound authentication refers to the configuration that determines the type of accepted authentication for inbound requests. This authentication is advertised in the interoperable object reference (IOR) that the client retrieves from the name server.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the settings to specify for CSI inbound communication.

The configureCSIInbound command configures various settings for CSI inbound communication. Review the following list of optional parameters to determine the attributes to set in your configuration:

Parameter	Description
-securityDomainName	Specifies the name of the security configuration. If you do not specify a security domain name, the command modifies the global security configuration. (String)
-messageLevelAuth	Specifies whether clients connecting to this server must specify a user ID and password. Specify `Never` to disable the user ID and password requirement. Specify `Supported` to accept a user ID and password. Specify `Required` to require a user ID and password. (String)
-supportedAuthMechList	Specifies the authentication mechanism to use. Specify `KRB5` for Kerberos authentication, `LTPA` for Lightweight Third-Party Authentication, `BasicAuth` for basic authentication, and `custom` to use your own authentication token implementation. You can specify more than one, separated by the pipe character (|). (String)
-clientCertAuth	Specifies whether a client that connects to the server must connect using an SSL certificate. Specify `Never` to allow clients to connect without SSL certificates. Specify `Supported` to accept clients connecting with and without SSL certificates. Specify `Required` to require clients to use SSL certificate. (String)
-transportLayer	Specifies the transport layer support level. Specify `Never` to disable transport layer support. Specify `Supported` to enable transport layer support. Specify `Required` to require transport layer support. (String)
-sslConfiguration	Specifies the SSL configuration alias to use for inbound transport. (String)
-enableIdentityAssertion	Specifies whether to enable identity assertion. When using the identity assertion authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element. Specify `true` for the -enableIdentityAssertion parameter to enable identity assertion. (Boolean)
-trustedIdentities	Specifies a list of trusted server identities, separated by the pipe character (|). To specify a null value, set the value of the -trustedIdentities parameter as an empty string (`""`). (String)
-statefulSession	Specifies whether to enable a stateful session. Specify `true` to enable a stateful session. (Boolean)
-enableAttributePropagation	Specifies whether to enable security attribute propagation. Security attribute propagation allows the application server to transport authenticated subject contents and security context information from one server to another in your configuration. Specify `true` to enable security attribute propagation. (Boolean)

3. Configure CSI inbound communication authentication.

   The configureCSIInbound command configures the CSIv2 Inbound authentication on a security domain or on the global security configuration. When configuring CSI Inbound in a security domain for the first time, the CSI objects are copied from global security. Then, the changes are applied to configuration.

   Use the configureCSIInbound command to configure CSI inbound authentication for a security domain or the global security configuration, as the following Jython example demonstrates:

```
AdminTask.configureCSIInbound('-securityDomainName testDomain -messageLevelAuth Supported
-supportedAuthMechList KRB5|LTPA -clientCertAuth Supported -statefulSession true')
```

4. Save your configuration changes.

- Configure CSI outbound communication authentication.

  Outbound authentication refers to the configuration that determines the type of authentication that is performed for outbound requests to downstream servers.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the settings to specify for CSI outbound communication.

     The configureCSIOutbound command configures various settings for CSI outbound communication. Review the following list of optional parameters to determine the attributes to set in your configuration:

Parameter	Description
-securityDomainName	Specifies the name of the security configuration. If you do not specify a security domain name, the command modifies the global security configuration. (String)
-enableAttributePropagation	Specifies whether to enable security attribute propagation. Security attribute propagation allows the application server to transport authenticated subject contents and security context information from one server to another in your configuration. Specify `true` to enable security attribute propagation. (Boolean)
-enableIdentityAssertion	Specifies whether to enable identity assertion. When using the identity assertion authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element. Specify `true` for the -enableIdentityAssertion parameter to enable identity assertion. (Boolean)
-useServerIdentity	Specifies whether to use the server identity to establish trust with the target server. Specify `true` to use the server identity. (Boolean)

Parameter	Description
-trustedId	Specifies the trusted identity that the application server uses to establish trust with the target server. (String)
-trustedIdentityPassword	Specifies the password of the trusted server identity. (String)
-messageLevelAuth	Specifies whether clients connecting to this server must specify a user ID and password. Specify `Never` to disable the user ID and password requirement. Specify `Supported` to accept a user ID and password. Specify `Required` to require a user ID and password. (String)
-supportedAuthMechList	Specifies the authentication mechanism to use. Specify `KRB5` for Kerberos authentication, `LTPA` for Lightweight Third-Party Authentication, `BasicAuth` for basic authentication, and `custom` to use your own authentication token implementation. You can specify more than one, separated by the pipe character (|). (String)
-clientCertAuth	Specifies whether a client that connects to the server must connect using an SSL certificate. Specify `Never` to allow clients to connect without SSL certificates. Specify `Supported` to accept clients connecting with and without SSL certificates. Specify `Required` to require clients to use SSL certificate. (String)
-transportLayer	Specifies the transport layer support level. Specify `Never` to disable transport layer support. Specify `Supported` to enable transport layer support. Specify `Required` to require transport layer support. (String)
-sslConfiguration	Specifies the SSL configuration alias to use for inbound transport. (String)
-statefulSession	Specifies whether to enable a stateful session. Specify `true` to enable a stateful session. (Boolean)
-enableOutboundMapping	Specifies whether to enable custom outbound identity mapping. Specify `true` to enable custom outbound identity mapping. (Boolean)
-trustedTargetRealms	Specifies a list of target realms to trust. Separate each realm name with the pipe character (|). (String)

3. Configure CSI outbound communication authentication.

   The configureCSIOutbound command configures the CSIv2 outbound authentication in a security domain or in the global security configuration. When configuring CSI outbound authentication in a security domain for the first time, the application server copies the CSI objects from global security. Then, the application server applies the changes to that configuration.

   Use the configureCSIOutbound command to configure CSI outbound authentication for a security domain or the global security configuration, as the following Jython example demonstrates:

```
AdminTask.configureCSIOutbound('-securityDomainName testDomain -enableIdentityAssertion true
-trustedId myID -trustedIdentityPassword myPassword123 -messageLevelAuth Required
-trustedTargetRealms realm1|realm2|realm3')
```

4. Save your configuration changes.

Configuring trust association using scripting

Use the wsadmin tool to configure and manage trust association configurations in a multiple security domain environment. Trust association enables the integration of the application server security and third-party security servers. More specifically, a reverse proxy server can act as a front-end authentication server while the product applies its own authorization policy onto the resulting credentials that are passed by the proxy server.

Before you begin

You must meet the following requirements before configuring a trust association:

- You must have the administrator or new admin role.
- Enable global security in your environment.
- Configure multiple realms using security domains in your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure a trust association.

   Use the configureTrustAssociation command to enable the trust association. You can also use this command to create or modify a trust association interceptor.

   The following Jython command creates a trust association for the `testDomain` security domain and configures the trust association to act as a reverse proxy server:

```
AdminTask.configureTrustAssociation('-securityDomainName testDomain -enable true')
```

3. Configure the trust association interceptor.

Use the configureInterceptor command to modify an existing interceptor. The following Jython command uses a WebSEAL interceptor to configure single sign-on for the `testDomain` security domain:

```
AdminTask.configureInterceptor('[-interceptor com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus
-securityDomainName testDomain -customProperties ["com.ibm.websphere.security.trustassociation.types=webseal",
"com.ibm.websphere.security.webseal.loginId=websealLoginID","com.ibm.websphere.security.webseal.id=iv-user"]]')
```

4. Save your configuration changes.

Mapping resources to security domains using scripting

Use this topic to assign management resources to security domains. Set management resources to your security domains to customize your security configuration for a cell, server, or cluster.

Before you begin

Users assigned to the administrator role can configure security domains. Verify that you have the appropriate administrative role before configuring security domains. Also, create a security domain, or copy an existing security domain before assigning resources to a security domain.

About this task

After creating a security domain, you can map management resources to the security domain. You can assign resources to a security domain at the server, cell, and cluster level. Use the following steps to assign a resource to a security domain:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine which security domain to map a resource.

   Use the listSecurityDomains command to view a list of security domains in your configuration. Specify `true` for the optional -listDescription parameter to list the description for each security domain, as the following Jython example demonstrates:

```
print AdminTask.listSecurityDomains('-listDescription true')
```

   The command returns the following example attribute list output:

```
{{name myDomain}
{description {security domain for administrative applications}}}
{{name domain2}
{description {new domain for cell123}}}
```

3. Assign a resource to a security domain.

   Use the mapResourceToSecurityDomain command to assign a management resource to the security domain. For example, use the following Jython command to secure all applications on the `server1` cell with the security attributes in the `domain2` security domain:

```
AdminTask.mapResourceToSecurityDomain('-securityDomainName domain2 -resourceName Cell=myCell:Node=myNode:Server=server1')
```

4. Save your configuration changes.

Results

Your security domain is updated in your configuration. All applications in the specified resource use the security attributes specified by the security domain. If the security domain does not contain all security attributes, then the missing attributes are obtained from the global security configuration.

What to do next

Restart each resource that you assigned to a security domain.

Removing resources from security domains using scripting

Use this topic to remove management resources from security domains. Remove all resources from a security domain before deleting the security domain from your configuration.

Before you begin

Users assigned to the administrator role can configure security domains. Verify that you have the appropriate administrative role before configuring security domains.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the security domain to edit.

   Use the listSecurityDomains command to view a list of security domains in your configuration. Specify `true` for the optional -listDescription parameter to list the description for each security domain, as the following Jython example demonstrates:

```
print AdminTask.listSecurityDomains('-listDescription true')
```

   The command returns the following example output:

```
myDomain - security domain for administrative applications
domain2 - new domain for cell123
```

3. Verify the management resources that are assigned to the security domain.

   Use the listResourcesInSecurityDomain command to view a list of resources that are mapped to the security domain of interest, as the following Jython example demonstrates:

```
print AdminTask.listResourcesInSecurityDomain('-securityDomainName domain2')
```

   The command returns the following example output:

```
"Cell=myhostCell01"
```

4. Remove the resource from the security domain.

   Use the removeResourceFromSecurityDomain command to remove a management resource from the security domain. For example, use the following Jython command to remove the `Cell01` cell resource from the `domain2` security domain:

```
AdminTask.removeResourceFromSecurityDomain('-securityDomainName domain2 -resourceName Cell=myhostCell01')
```

5. Save your configuration changes.

What to do next

Restart each management resource that you removed from a security domain.

Removing security domains using scripting

Use this topic to delete security domains from your configuration using the wsadmin tool. Remove security domains that are not needed in your security configuration.

Before you begin

Users assigned to the administrator role can configure security domains. Verify that you have the appropriate administrative role before configuring security domains. A security domain must exist in your configuration.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the security domain to delete.

   Use the listSecurityDomains to view a list of security domains in your configuration. Specify `true` for the optional -listDescription parameter to list the description for each security domain, as the following Jython example demonstrates:

```
print AdminTask.listSecurityDomains('-listDescription true')
```

   The command returns the following example output:

```
{{name myDomain}
{description {security domain for administrative applications}}}
{{name domain2}
{description {new domain for cell123}}}
```

3. Verify that no resources are assigned to the security domain to delete.

   You can use this step to manually remove resources from the security domain of interest. You do not need to complete this step if you want to delete the security domain and each assigned resource. Use

the listResourcesInSecurityDomain command to view a list of resources that are mapped to the security domain of interest, as the following Jython example demonstrates:

```
print AdminTask.listResourcesInSecurityDomain('-securityDomainName domain2')
```

If the command returns the name of a resource, use the removeResourceFromSecurityDomain command to remove a resource from the security domain. For example, use the following Jython command to remove the Cell01 cell resource from the domain2 security domain:

```
"AdminTask.removeResourceFromSecurityDomain('-securityDomainName domain2 -resourceName Cell=myhostCell01')"
```

4. Delete the security domain from your configuration.

   Use the deleteSecurityDomain command to delete the security domain. If a resource associated with the domain was deleted from the system but the mapping was not removed form the domain, specify the optional -force parameter to remove the domain, as the following Jython example demonstrates:

```
AdminTask.deleteSecurityDomain('-securityDomainName domain2 -force true')
```

5. Save your configuration changes.

Removing user registries using scripting

You can use the wsadmin tool to remove user registries from global security or security domain configurations. Use the steps in this topic to remove Lightweight Directory Access Protocol (LDAP), local operating system, custom, or federated repository user registries from your global security or security domain configurations.

Before you begin

You must meet the following requirements before configuring local operating system user registries:

- You must have the administrator or new admin role.
- Enable global security in your environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the registry to remove.

   Use the getUserRegistryInfo command to display information about a user registry from the global security configuration or in a security domain. You must specify the type of user registry of interest. Valid values are LDAPUserRegistry, WIMUserRegistry, CustomUserRegistry, and LocalOSUserRegistry. The following command returns a list of values in the local operating system user registry object for the domain2 security domain, as the following example Jython demonstrates:

```
AdminTask.getUserRegistryInfo('-securityDomainName domain2 -userRegistryType LocalOSUserRegistry')
```

3. Determine whether the registry of interest is the active user registry.

   You cannot remove the active user registry. Use the getActiveSecuritySettings command to see check if the user registry is the active user registry before removing it.

4. Remove the registry of interest.

   Use the unconfigureUserRegistry command to remove the registry of interest. If you remove the user registry from the global security configuration, then the command reduces the registry object to the minimum values for the configuration. If you remove the user registry from a security domain, then the command removes the configuration object from the security domain. The following Jython example removes the local operating system user registry configuration from the domain2 security domain:

```
AdminTask.unconfigureUserRegistry('-securityDomainName domain2 -userRegistryType LocalOSUserRegistry')
```

5. Save your configuration changes.

SecurityDomainCommands command group for the AdminTask object

You can use the Jython scripting language to configure and administer security domains with the wsadmin tool. Use the commands and parameters in the SecurityDomainCommands group to create and manage security domains, assign servers and clusters to security domains as resources, and to query the security domain configuration.

Use the following commands to administer the security domain configuration:

- copySecurityDomain
- copySecurityDomainFromGlobalSecurity
- createSecurityDomain
- deleteSecurityDomain
- getSecurityDomainForResource
- listResourcesInSecurityDomain
- listSecurityDomains
- “listSecurityDomainsForResources” on page 570
- mapResourceToSecurityDomain
- modifySecurityDomain
- removeResourceFromSecurityDomain

copySecurityDomain

The copySecurityDomain command creates a new security domain by copying an existing security domain. If the security configuration defines an active user registry, provide a realm name for the newly create security domain. If you do not specify a realm name, the system creates a realm name.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the new security domain that the system creates by copying another security domain. (String)

-copyFromSecurityDomainName
Specifies the name of the existing security domain that the system uses to create the new security domain. (String)

Optional parameters

-securityDomainDescription
Specifies a description for the new security domain. (String)

-realmName
Specifies the name of the realm in the new security domain. The system creates a name for the realm if you do not specify a value for this parameter. (String)

Return value

The command returns the configuration ID of the new security domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.copySecurityDomain('-securityDomainName copyOfDomain2 -copyFromSecurityDomainName Domain2')
```

- Using Jython list:

```
AdminTask.copySecurityDomain('-securityDomainName', 'copyOfDomain2', '-copyFromSecurityDomainName', 'Domain2')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.copySecurityDomain('-interactive')
```

copySecurityDomainFromGlobalSecurity

The copySecurityDomainFromGlobalSecurity command creates a security domain by copying the global security configuration. If an active user registry exists for the global security configuration, provide a realm name for the newly created security domain. If you do not specify a realm name, then the system creates a realm name.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the new security domain that the system copies from the global security configuration. (String)

Optional parameters

-securityDomainDescription
Specifies a description for the new security domain. (String)

-realmName
Specifies the name of the realm in the new security configuration. The system creates a name for the realm if you do not specify a value for the -realmName parameter. (String)

Return value

The command returns the configuration ID of the new security domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.copySecurityDomainFromGlobalSecurity('-securityDomainName GSCopy -securityDomainDescription
"copy of global security" -realmName myRealm')
```

- Using Jython list:

```
AdminTask.copySecurityDomainFromGlobalSecurity('-securityDomainName', 'GSCopy', '-securityDomainDescription',
'"copy of global security"', '-realmName myRealm')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.copySecurityDomainFromGlobalSecurity('-interactive')
```

createSecurityDomain

The createSecurityDomain command creates the security domain-security.xml and domain-security-map.xml files under the profile_root/config/cells/*cellName*/securityDomain/*configurationName* directory. The system creates an empty domain-security.xml file.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the new security domain to create. (String)

Optional parameters

-securityDomainDescription
Specifies a description of the new security domain. (String)

Return value

The command returns the configuration ID of the new security domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.createSecurityDomain('-securityDomainName newDomain -securityDomainDescription "new security domain"')
```

- Using Jython list:

```
AdminTask.createSecurityDomain('-securityDomainName', 'newDomain', '-securityDomainDescription',
'"new security domain"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createSecurityDomain('-interactive')
```

deleteSecurityDomain

The deleteSecurityDomain command removes the domain-security.xml and domain-security-map.xml files from the security domain directory. The command returns an error if resources are mapped to the security domain of interest. To delete the security domain when resources are mapped to the security domain of interest, specify the value for the -force parameter as `true`.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security domain to delete. (String)

Optional parameters

-force
Specifies that the system deletes the security domain without checking for resources that are associated with the domain. Use this option when the resources in the security domains are not valid resources. The default value for the -force parameter is `false`. (Boolean)

Return value

The command does not return output if the system successfully removes the security domain configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteSecurityDomain('-securityDomainName mySecurityDomain -force true')
```

- Using Jython list:

```
AdminTask.deleteSecurityDomain('-securityDomainName', 'mySecurityDomain', '-force', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteSecurityDomain('-interactive')
```

getSecurityDomainForResource

The getSecurityDomainForResource command displays the security domain for a specific resource. If the resource is not mapped to a domain, the command does not return output.

Target object

None.

Required parameters

-resourceName
Specifies the name of the resource of interest. Specify the value in the following format: `Cell=:Node=`*myNode*`:Server=`*myServer* (String)

Optional parameters

-getEffectiveDomain
Specifies whether the command returns the effective domain of the resource if the resource is not directly mapped to a domain. The default value is `true`. Specify `false` if you do not want to display the effective domain if the resource is not directly mapped to a domain. (Boolean)

Return value

The command returns the security domain name as a string.

Batch mode example usage

- Using Jython string:

```
AdminTask.getSecurityDomainForResource('-resourceName Cell=:Node=myNode:Server=myServer')
```

- Using Jython list:

```
AdminTask.getSecurityDomainForResource('-resourceName', 'Cell=:Node=myNode:Server=myServer')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getSecurityDomainForResource('-interactive')
```

listResourcesInSecurityDomain

The listResourcesInSecurityDomain command displays the servers or clusters that are associated with a specific security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security domain of interest. (String)

-expandCell
Specifies whether to display the servers in the cell. Specify `true` to display the specific servers, or specify `false` to list the cell information only. (Boolean)

Return value

The command returns an array that contains the names of the resources that are mapped to the security domain of interest in the format: `Cell=<cell name>:Node=<node name>:Server=<server name>`.

Batch mode example usage

- Using Jython string:

```
AdminTask.listResourcesInSecurityDomain('-securityDomainName myDomain')
```

- Using Jython list:

```
AdminTask.listResourcesInSecurityDomain('-securityDomainName', 'myDomain')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listResourcesInSecurityDomain('-interactive')
```

listSecurityDomains

The listSecurityDomains command lists each security domain configured for the server.

Target object

None.

Optional parameters

-listDescription
Specifies whether to display the description of the security domains. Specify `true` to display the descriptions of the security domains. (Boolean)

-doNotDisplaySpecialDomains
Specifies whether to exclude special domains. Specify `true` to exclude the special domains in the command output, or `false` to display the special domains. (Boolean)

Return value

The command returns an array that contains the names of security domains that are configured for the server. The command returns an array of attribute lists that contain the name and description for each security domain if the -listDescription parameter is specified.

Batch mode example usage

- Using Jython string:

```
AdminTask.listSecurityDomains('-listDescription true')
```

- Using Jython list:

```
AdminTask.listSecurityDomains('-listDescription', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSecurityDomains('-interactive')
```

listSecurityDomainsForResources

The listSecurityDomainsForResources command lists the security domains that are associated with the resources of interest.

Target object

None.

Required parameters

-resourceNames
Specifies one or more resources for which the command returns the associated security domains. Specify each resource separated by the plus sign character (+). (String)

Return value

The command returns the list of resources specified by the -resourceNames parameter and the security domains to which each resource is mapped.

Batch mode example usage

- Using Jython string:

```
AdminTask.listSecurityDomainsForResources('-resourceNames resource1+resource2+resource3')
```

- Using Jython list:

```
AdminTask.listSecurityDomainsForResources('-resourceNames', 'resource1+resource2+resource3')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSecurityDomainsForResources('-interactive')
```

mapResourceToSecurityDomain

The mapResourceToSecurityDomain command maps a resource to a security domain. The system adds an entry for each resource to the domain-security-map.xml file.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security domain of interest. (String)

-resourceName
Specifies the name of the resource to which the system maps the security domain of interest. Specify the value in the following format: `Cell=:Node=myNode:Server=myServer` (String)

Return value

The command does not return output if the system successfully assigns the resource to the security domain of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.mapResourceToSecurityDomain('-securityDomainName mySecurityDomain -resourceName
-resourceName Cell=:Node=myNode:Server=myServer')
```

- Using Jython list:

```
AdminTask.mapResourceToSecurityDomain('-securityDomainName', 'mySecurityDomain', '-resourceName',
'-resourceName Cell=:Node=myNode:Server=myServer')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.mapResourceToSecurityDomain('-interactive')
```

modifySecurityDomain

The modifySecurityDomain command changes the description of a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security domain to edit. (String)

Optional parameters

-securityDomainDescription
Specifies the new description for the security domain of interest. (String)

Return value

The command does not return output if the system successfully modifies the security domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifySecurityDomain('-securityDomainName myDomain -securityDomainDescription
"my new description"')
```

- Using Jython list:

```
AdminTask.modifySecurityDomain('-securityDomainName', 'myDomain', '-securityDomainDescription',=
'"my new description"')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifySecurityDomain('-interactive')
```

removeResourceFromSecurityDomain

The removeResourceFromSecurityDomain command removes a resource from a security domain mapping. The command removes the resource entry from the domain-security-map.xml file.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security domain from which to remove the resource. (String)

-resourceName
Specifies the name of the resource to remove. Specify the value in the following format: `Cell=:Node=myNode:Server=myServer` (String)

Return value

The command does not return output if the system successfully removes the resource from the security domain.

Batch mode example usage

- Using Jython string:

```
AdminTask.removeResourceFromSecurityDomain('-securityDomainName myDomain -resourceName
Cell=:Node=myNode:Server=myServer')
```

- Using Jython list:

```
AdminTask.removeResourceFromSecurityDomain('-securityDomainName', 'myDomain', '-resourceName',
'Cell=:Node=myNode:Server=myServer')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeResourceFromSecurityDomain('-interactive')
```

SecurityConfigurationCommands command group for the AdminTask object

You can use the Jython scripting language to configure security with the wsadmin tool. Use the commands and parameters in the SecurityConfigurationCommands group to configure and manage user registries, single sign-on, data entries, trust association, login modules, and interceptors.

Use the following command to administer user registry configurations:

- configureAdminCustomUserRegistry
- configureAdminLDAPUserRegistry
- configureAdminLocalOSUserRegistry
- configureAdminWIMUserRegistry
- configureAppCustomUserRegistry
- configureAppLDAPUserRegistry
- configureAppLocalOSUserRegistry
- configureAppWIMUserRegistry
- getLTPATimeout
- setLTPATimeout
- getUserRegistryInfo
- unconfigureUserRegistry

Use the following commands to administer Java Authentication and Authorization Service (JAAS) login configurations:

- configureLoginEntry
- configureLoginModule
- getJAASLoginEntryInfo
- listJAASLoginEntries
- listLoginModules
- unconfigureJAASLoginEntry
- unconfigureLoginModule

Use the following commands to administer data entry configurations:

- createAuthDataEntry
- deleteAuthDataEntry
- getAuthDataEntry
- listAuthDataEntries
- modifyAuthDataEntry

Use the following commands to administer Common Secure Interoperability Version 2 (CSIv2) configurations:

- configureCSIInbound
- configureCSIOutbound
- getCSIInboundInfo
- getCSIOutboundInfo
- unconfigureCSIInbound
- unconfigureCSIOutbound

Use the following commands to administer trust association configurations:

- configureInterceptor
- configureTrustAssociation
- getTrustAssociationInfo
- listInterceptors
- unconfigureInterceptor
- unconfigureTrustAssociation

Use the following commands to manage your security configuration:

- configureAuthzConfig
- configureSingleSignon
- getActiveSecuritySettings
- “getAuthzConfigInfo” on page 607
- “getSingleSignon” on page 608
- setAdminActiveSecuritySettings
- setAppActiveSecuritySettings
- unconfigureAuthzConfig
- unsetAppActiveSecuritySettings

configureAdminCustomUserRegistry

The configureAdminCustomUserRegistry command configures a custom user registry in the global security configuration.

Target object

None.

Optional parameters

-autoGenerateServerId
Specifies whether the command automatically generates the server identity that the system uses for internal process communication. Specify `true` to automatically generate the server identity. (Boolean)

-serverId
Specifies the server identity in the repository that the system uses for internal process communication. (String)

-serverIdPassword
Specifies the password that corresponds to the server identity. (String)

-primaryAdminId
Specifies the name of the user with administrative privileges that is defined in the registry. This parameter does not apply to security configurations. (String)

-customRegClass
Specifies the class name that implements the UserRegistry interface in com.ibm.websphere.security property. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAdminCustomUserRegistry('-autoGenerateServerId true -serverIdPassword password4server
-primaryAdminId serverAdmin')
```

- Using Jython list:

```
AdminTask.configureAdminCustomUserRegistry(['autoGenerateServerId', 'true', '-serverIdPassword', 'password4server',
'-primaryAdminId', 'serverAdmin'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAdminCustomUserRegistry('-interactive')
```

configureAdminLDAPUserRegistry

The configureAdminLDAPUserRegistry command configures a Lightweight Directory Access Protocol (LDAP) user registry in the global security configuration.

Target object

None.

Optional parameters

-autoGenerateServerId
Specifies whether the command automatically generates the server identity used for internal process communication. Specify `true` to automatically generate the server identity. (Boolean)

-serverId
Specifies the server identity in the repository that the system uses for internal process communication. (String)

-serverIdPassword
Specifies the password that corresponds to the server identity. (String)

-primaryAdminId
Specifies the name of the user with administrative privileges that is defined in the registry. This parameter does not apply to security configurations. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-ldapServerType
Specifies the type of LDAP server. The default type is `IBM_DIRECTORY_SERVER`. (String)

Specify one of the following valid values:

- `IBM_DIRECTORY_SERVER`
- `IPLANET`
- `NETSCAPE`
- `NDS`
- `DOMINO502`
- `SECUREWAY`
- `ACTIVE_DIRECTORY`
- `CUSTOM`

-ldapHost
Specifies the host name of the LDAP server. (String)

-ldapPort
Specifies the port that the system uses to access the LDAP server. The default value is `389`. (String)

-baseDN
Specifies the base distinguished name (DN) of the directory service, which indicates the starting point for LDAP searches of the directory service. In most cases, bind DN and bind password are needed. However, when anonymous bind can satisfy all of the required functions, bind DN and bind password are not needed. (String)

-bindDN
Specifies the distinguished name for the application server, which is used to bind to the directory service. (String)

-bindPassword
Specifies the binding DN password for the LDAP server. (String)

-searchTimeout
Specifies the timeout value in seconds for an LDAP server to respond before stopping a request. The default value is `120` seconds. (Long)

-reuseConnection
Specifies whether the server reuses the LDAP connection. By default, this option is enabled. Specify `false` for this parameter only in rare situations where a router is used to distribute requests to multiple LDAP servers and when the router does not support affinity. (Boolean)

Note: When you disable the reuse of the LDAP connection, the application server creates a new LDAP connection for every LDAP search request. This situation impacts system performance if your environment requires extensive LDAP calls. This option is provided because the router is not sending the request to the same LDAP server. The option is also used when the idle connection timeout value or firewall timeout value between the application server and LDAP is too small.

-userFilter
Specifies the LDAP filter clause that the system uses to search the user registry for users. The default value is the default user filter for the LDAP server type. (String)

-groupFilter
Specifies the LDAP filter clause that the system uses to search the user registry for groups. The default value is the default group filter for the LDAP server type. (String)

-userIdMap
Specifies the LDAP filter that maps the short name of a user to an LDAP entry. The default value is the default user filter for the LDAP server type. (String)

-groupIdMap
Specifies the LDAP filter that maps the short name of a group to an LDAP entry. The default value is the default group filter for the LDAP server type. (String)

-groupMemberIdMap
Specifies the LDAP filter that identifies users to group memberships. (String)

-certificateMapMode
Specifies whether to map X.509 certificates into an LDAP directory by `EXACT_DN` or `CERTIFICATE_FILTER`. Specify `CERTIFICATE_FILTER` to use the specified certificate filter for the mapping. (String)

-certificateFilter
Specifies the filter certificate mapping property for the LDAP filter. The filter is used to map attributes in the client certificate to entries in the LDAP registry. (String)

The syntax or structure of this filter is: `(&(uid=${SubjectCN})(objectclass=inetOrgPerson))`. The left side of the filter specification is an LDAP attribute that depends on the schema that your LDAP server is configured to use. The right side of the filter specification is one of the public attributes in your client certificate. The right side must begin with a dollar sign ($) and open bracket ({) and end with a close bracket (}). You can use the following certificate attribute values on the right side of the filter specification. The case of the strings is important:

- ${UniqueKey}
- ${PublicKey}
- ${Issuer}
- ${NotAfter}
- ${NotBefore}
- ${SerialNumber}
- ${SigAlgName}
- ${SigAlgOID}
- ${SigAlgParams}
- ${SubjectCN}
- ${Version}

-krbUserFilter
Specifies the default value is the default user filter for the LDAP server type. (String)

-nestedGroupSearch
Specifies whether to perform a recursive nested group search. Specify `true` to perform a recursive nested group search, or specify `false` to disable recursive nested group searching. (Boolean)

-sslEnabled
Specifies whether to enable Secure Sockets Layer (SSL). Specify `true` to enable an SSL connection to the LDAP server. (Boolean)

-sslConfig
Specifies the SSL configuration alias to use for the secure LDAP connection. (String)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: "attr1=value1","attr2=value2" (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAdminCustomUserRegistry('-autoGenerateServerId true -serverIdPassword password4server
-primaryAdminId serverAdmin -ldapServerType NETSCAPE -ldapHost 195.168.1.1')
```

- Using Jython list:

```
AdminTask.configureAdminCustomUserRegistry(['-autoGenerateServerId', 'true', '-serverIdPassword', 'password4server',
'-primaryAdminId', 'serverAdmin', '-ldapServerType', 'NETSCAPE', '-ldapHost', '195.168.1.1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAdminLDAPUserRegistry('-interactive')
```

configureAdminLocalOSUserRegistry

The configureAdminLocalOSUserRegistry command configures a local operating system user registry in the global security configuration.

Target object

None.

Optional parameters

-autoGenerateServerId
Specifies whether the command automatically generates the server identity used for internal process communication. Specify `true` to automatically generate the server identity. (Boolean)

-serverId
Specifies the server identity in the repository that the system uses for internal process communication. (String)

-serverIdPassword
Specifies the password that corresponds to the server identity. (String)

-primaryAdminId
Specifies the name of the user with administrative privileges that is defined in the registry. This parameter does not apply to security configurations. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: "attr1=value1","attr2=value2" (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAdminLocalOSUserRegistry('-autoGenerateServerId true -serverIdPassword password4server
-primaryAdminId serverAdmin')
```

- Using Jython list:

```
AdminTask.configureAdminLocalOSUserRegistry(['autoGenerateServerId', 'true', '-serverIdPassword', 'password4server',
'-primaryAdminId', 'serverAdmin'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAdminLocalOSUserRegistry('-interactive')
```

configureAdminWIMUserRegistry

The configureAdminWIMUserRegistry command configures a federated repository user registry in the administrative security configuration.

Target object

None.

Optional parameters

-autoGenerateServerId
Specifies whether the command automatically generates the server identity used for internal process communication. Specify `true` to automatically generate the server identity. (Boolean)

-serverId
Specifies the server identity in the repository that the system uses for internal process communication. (String)

-serverIdPassword
Specifies the password that corresponds to the server identity. (String)

-primaryAdminId
Specifies the name of the user with administrative privileges that is defined in the registry. This parameter does not apply to security configurations. (String)

-realmName
Specifies the realm of the user registry. The system automatically generates a realm name if you do not specify a value for the -realmName parameter. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAdminWIMUserRegistry('-autoGenerateServerId true -serverIdPassword password4server
-primaryAdminId serverAdmin')
```

- Using Jython list:

```
AdminTask.configureAdminWIMUserRegistry(['autoGenerateServerId', 'true', '-serverIdPassword', 'password4server',
'-primaryAdminId', 'serverAdmin'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAdminWIMUserRegistry('-interactive')
```

configureAppCustomUserRegistry

The configureAppCustomUserRegistry command configures a custom user registry in an application security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Optional parameters

-realmName
Specifies the realm of the user registry. The system automatically generates a realm name if you do not specify a value for the -realmName parameter. (String)

-customRegClass
Specifies the class name that implements the UserRegistry interface in com.ibm.websphere.security property. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAppCustomUserRegistry('-securityDomainName testDomain -realmName server_name.domain:port_number')
```

- Using Jython list:

```
AdminTask.configureAppCustomUserRegistry(['-securityDomainName', 'testDomain', '-realmName',
'server_name.domain:port_number'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAppCustomUserRegistry('-interactive')
```

configureAppLDAPUserRegistry

The configureAppLDAPUserRegistry command configures LDAP user registries in a security configuration or a global security configuration.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Optional parameters

-realmName
Specifies the realm of the user registry. The system automatically generates a realm name if you do not specify a value for the -realmName parameter. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-ldapServerType
Specifies the type of LDAP server. The default type is `IBM_DIRECTORY_SERVER`. (String)

Specify one of the following valid values:

- `IBM_DIRECTORY_SERVER`
- `IPLANET`
- `NETSCAPE`
- `NDS`
- `DOMINO502`
- `SECUREWAY`
- `ACTIVE_DIRECTORY`
- `CUSTOM`

-ldapHost
Specifies the host name of the LDAP server. (String)

-ldapPort
Specifies the port that the system uses to access the LDAP server. The default value is `389`. (String)

-baseDN
Specifies the base distinguished name (DN) of the directory service, which indicates the starting point for LDAP searches of the directory service. In most cases, bind DN and bind password are needed. However, when anonymous bind can satisfy all of the required functions, bind DN and bind password are not needed. (String)

-bindDN
Specifies the distinguished name for the application server, which is used to bind to the directory service. (String)

-bindPassword
Specifies the binding DN password for the LDAP server. (String)

-searchTimeout
Specifies the timeout value in seconds for an LDAP server to respond before stopping a request. The default value is `120` seconds. (Long Integer)

-reuseConnection
Specifies whether the server reuses the LDAP connection. By default, this option is enabled. Specify `false` for this parameter only in rare situations where a router is used to distribute requests to multiple LDAP servers and when the router does not support affinity. (Boolean)

Note: When you disable the reuse of the LDAP connection, the application server creates a new LDAP connection for every LDAP search request. This situation impacts system performance if your environment requires extensive LDAP calls. This option is provided because the router is not sending the request to the same LDAP server. The option is also used when the idle connection timeout value or firewall timeout value between the application server and LDAP is too small.

-userFilter
Specifies the LDAP filter clause that the system uses to search the user registry for users. The default value is the default user filter for the LDAP server type. (String)

-groupFilter
Specifies the LDAP filter clause that the system uses to search the user registry for groups. The default value is the default group filter for the LDAP server type. (String)

-userIdMap
Specifies the LDAP filter that maps the short name of a user to an LDAP entry. The default value is the default user filter for the LDAP server type. (String)

-groupIdMap
Specifies the LDAP filter that maps the short name of a group to an LDAP entry. The default value is the default group filter for the LDAP server type. (String)

-groupMemberIdMap
Specifies the LDAP filter that identifies users to group memberships. (String)

-certificateMapMode
Specifies whether to map X.509 certificates into an LDAP directory by `EXACT_DN` or `CERTIFICATE_FILTER`. Specify `CERTIFICATE_FILTER` to use the specified certificate filter for the mapping. (String)

-certificateFilter
Specifies the filter certificate mapping property for the LDAP filter. The filter is used to map attributes in the client certificate to entries in the LDAP registry. (String)

The syntax or structure of this filter is: `(&(uid=${SubjectCN})(objectclass=inetOrgPerson))`. The left side of the filter specification is an LDAP attribute that depends on the schema that your LDAP server is configured to use. The right side of the filter specification is one of the public attributes in your client certificate. The right side must begin with a dollar sign ($) and open bracket ({) and end with a close bracket (}). You can use the following certificate attribute values on the right side of the filter specification. The case of the strings is important:

- ${UniqueKey}
- ${PublicKey}
- ${Issuer}

- ${NotAfter}
- ${NotBefore}
- ${SerialNumber}
- ${SigAlgName}
- ${SigAlgOID}
- ${SigAlgParams}
- ${SubjectCN}
- ${Version}

-krbUserFilter
Specifies the default value is the default user filter for the LDAP server type. (String)

-nestedGroupSearch
Specifies whether to perform a recursive nested group search. Specify `true` to perform a recursive nested group search, or specify `false` to disable recursive nested group searching. (Boolean)

-sslEnabled
Specifies whether to enable Secure Sockets Layer (SSL). Specify `true` to enable an SSL connection to the LDAP server. (Boolean)

-sslConfig
Specifies the SSL configuration alias to use for the secure LDAP connection. (String)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAppLDAPUserRegistry('-securityDomainName testDomain -ldapServerType NETSCAPE -ldapHost
195.168.1.1 -searchTimeout 300')
```

- Using Jython list:

```
AdminTask.configureAppLDAPUserRegistry(['-securityDomainName', 'testDomain', '-ldapServerType',
'NETSCAPE', '-ldapHost', '195.168.1.1', '-searchTimeout', '300'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAppLDAPUserRegistry('-interactive')
```

configureAppLocalOSUserRegistry

The configureAppLocalOSUserRegistry command configures a local operating system user registry in a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Optional parameters

-realmName
Specifies the realm of the user registry. The system automatically generates a realm name if you do not specify a value for the -realmName parameter. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAppLocalOSUserRegistry('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.configureAppLocalOSUserRegistry(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAppLocalOSUserRegistry('-interactive')
```

configureAppWIMUserRegistry

The configureAppWIMUserRegistry command configures federated repository user registries in a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Optional parameters

-realmName
Specifies the realm of the user registry. The system automatically generates a realm name if you do not specify a value for the -realmName parameter. (String)

-verifyRegistry
Specifies whether to verify that the user registry configuration is correct. If you set this parameter to `true`, then the system verifies the registry by making a call to the user registry to verify the admin ID. If you specify a server ID and password, then the system verifies the user and password with the user registry. Set the parameter to `false` to store the attributes in the configuration without validation. The command verifies the registry configuration by default. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: "attr1=value1","attr2=value2" (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAppWIMUserRegistry('-securityDomainName testDomain -realmName testRealm')
```

- Using Jython list:

```
AdminTask.configureAppWIMUserRegistry(['securityDomainName', 'testDomain', '-realmName', 'testRealm'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAppWIMUserRegistry('-interactive')
```

getLTPATimeout

The getLTPATimeout command displays the number of seconds that the system waits before the LTPA request reaches timeout.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns the number of seconds that the server waits before the LTPA request is cancelled.

Batch mode example usage

- Using Jython string:

```
AdminTask.getLTPATimeout('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getLTPATimeout(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getLTPATimeout('-interactive')
```

setLTPATimeout

The setLTPATimeout command sets the amount of time that the system waits before the LTPA request becomes invalid.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-timeout
Specifies the amount of time, in seconds, before the request times out. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setLTPATimeout('-timeout 120')
```

- Using Jython list:

```
AdminTask.setLTPATimeout(['timeout', '120'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setLTPATimeout('-interactive')
```

getUserRegistryInfo

The getUserRegistryInfo command displays information about a user registry in a security domain or in the global security configuration. If you do not specify a value for the -userRegistryType parameter, the command returns the active user registry information.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-userRegistryType
Specifies the type of user registry. Specify `LDAPUserRegistry` for LDAP user registries. Specify `WIMUserRegistry` for federated repository user registries. Specify `CustomUserRegistry` for custom user registries. Specify `LocalOSUserRegisty` for local operating system user registries. (String)

Return value

The command returns configuration information in the form of attribute and value pairs for the user registry object of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getUserRegistryInfo('-securityDomainName testDomain -userRegistryType LDAPUserRegistry')
```

- Using Jython list:

```
AdminTask.getUserRegistryInfo(['securityDomainName', 'testDomain', '-userRegistryType', 'LDAPUserRegistry'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getUserRegistryInfo('-interactive')
```

unconfigureUserRegistry

The unconfigureUserRegistry command modifies the user registry. For a global security configuration, the command reduces the user registry to the minimum registry values. For application-level security, the command removes the user registry from the security domain of interest.

Target object

None.

Required parameters

-userRegistryType
Specifies the type of user registry. Specify `LDAPUserRegistry` for LDAP user registries. Specify `WIMUserRegistry` for federated repository user registries. Specify `CustomUserRegistry` for custom user registries. Specify `LocalOSUserRegisty` for local operating system user registries. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureUserRegistry('-userRegistryType WIMUserRegistry -securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureUserRegistry(['userRegistryType', 'WIMUserRegistry', '-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureUserRegistry('-interactive')
```

configureLoginEntry

The configureLoginEntry command configures a Java Authentication and Authorization Service (JAAS) login entry in a security domain or in the global security configuration. You can use this command to modify existing JAAS login entries or to create new login entries.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. If you do not specify a security domain name, the system updates the global security configuration. (String)

-loginModules
Specifies a comma (**,**) separated list of login module class names. Specify the list in the order that the system calls them. (String)

-authStrategies
Specifies a comma-separated list of authentication strategies that sets the authentication behavior as authentication proceeds down the list of login modules. You must specify one authentication strategy for each login module. (String)

Specify one or many of the following values in a comma (**,**) separated list:

- `REQUIRED`

  Specifies that the LoginModule module is required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list for each realm.
- `REQUISITE`

  Specifies that the LoginModule module is required to succeed. If authentication is successful, the process continues down the LoginModule list in the realm entry. If authentication fails, control immediately returns to the application. Authentication does not proceed down the LoginModule list.
- `SUFFICIENT`

  Specifies that the LoginModule module is not required to succeed. If authentication succeeds, control immediately returns to the application. Authentication does not proceed down the LoginModule list. If authentication fails, the process continues down the list.
- `OPTIONAL`

  Specifies that the LoginModule module is not required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureLoginEntry('[-loginType application -loginEntryAlias JAASLoginEntry1 -authStrategies "
REQUIRED,REQUISITE"]')
```

- Using Jython list:

```
AdminTask.configureLoginEntry(['loginType', 'application', '-loginEntryAlias', 'JAASLoginEntry1', '-authStrategies',
'REQUIRED,REQUISITE'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureLoginEntry('-interactive')
```

configureLoginModule

The configureLoginModule command modifies an existing login module or creates a new login module on an existing JAAS login entry in the global security configuration or in a security domain.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

-loginModule
Specifies the name of the login module. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. (String)

-useLoginModuleProxy
Specifies that the JAAS loads the login module proxy class. JAAS then delegates calls to the login module classes that are defined in the Module class name field. Specify `true` to use the login module proxy. (Boolean)

-authStrategy
Specifies the authentication behavior as authentication proceeds down the list of login modules. (String)

Specify one of the following values:

- `REQUIRED`

  Specifies that the LoginModule module is required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list for each realm.

- `REQUISITE`

  Specifies that the LoginModule module is required to succeed. If authentication is successful, the process continues down the LoginModule list in the realm entry. If authentication fails, control immediately returns to the application. Authentication does not proceed down the LoginModule list.

- `SUFFICIENT`

  Specifies that the LoginModule module is not required to succeed. If authentication succeeds, control immediately returns to the application. Authentication does not proceed down the LoginModule list. If authentication fails, the process continues down the list.

- `OPTIONAL`

  Specifies that the LoginModule module is not required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list.

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: [`"attr1=value1","attr2=value2"`] (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureLoginModule('-loginType application -loginEntryAlias JAASLoginEntry1 -loginModule class1')
```

- Using Jython list:

```
AdminTask.configureLoginModule(['loginType', 'application', '-loginEntryAlias', 'JAASLoginEntry1', '-loginModule',
'class1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureLoginModule('-interactive')
```

getJAASLoginEntryInfo

The getJAASLoginEntryInfo command displays configuration for a specific JAAS login entry.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an attribute list that contains configuration information for the JAAS login entry of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getJAASLoginEntryInfo('-loginType application -loginEntryAlias JAASLoginEntry -securityDomainName
testDomain')
```

- Using Jython list:

```
AdminTask.getJAASLoginEntryInfo(['loginType', 'application', '-loginEntryAlias', 'JAASLoginEntry', '-securityDomainName',
'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getJAASLoginEntryInfo('-interactive')
```

listJAASLoginEntries

The listJAASLoginEntries command displays each defined JAAS login modules for given type in a security domain or the global security configuration.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an array of attribute lists that contain the login entries for the login type of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.listJAASLoginEntries('-loginType application -securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.listJAASLoginEntries(['loginType', 'application','-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listJAASLoginEntries('-interactive')
```

listLoginModules

The listLoginModules command displays the class names and associated options for a specific JAAS login module in a security domain or in the global security configuration.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an array that contains the login modules in a specific login entry.

Batch mode example usage

- Using Jython string:

```
AdminTask.listLoginModules('-loginType system -loginEntryAlias JAASLoginEntry')
```

- Using Jython list:

```
AdminTask.listLoginModules(['loginType', 'system', '-loginEntryAlias', 'JAASLoginEntry'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listLoginModules('-interactive')
```

unconfigureJAASLoginEntry

The unconfigureJAASLoginEntry command removes a JAAS login entry from the global security configuration or a security domain. You cannot remove all login entries. The command returns an error if it cannot remove the login entry of interest.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureJAASLoginEntry('-loginType application -loginEntryAlias myLoginEntry')
```

- Using Jython list:

```
AdminTask.unconfigureJAASLoginEntry(['loginType', 'application', '-loginEntryAlias', 'myLoginEntry'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureJAASLoginEntry('-interactive')
```

unconfigureLoginModule

The unconfigureLoginModule command removes a login module class from a login module entry.

Target object

None.

Required parameters

-loginType
Specifies the type of JAAS login entry of interest. Specify `system` for the system login type or `application` for the application login type. (String)

-loginEntryAlias
Specifies an alias that identifies the JAAS login entry in the configuration. (String)

-loginModule
Specifies the name of the login module class to remove from the configuration. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureLoginModule('-loginType system -loginEntryAlias systemLoginEntry -loginModule moduleClass')
```

- Using Jython list:

```
AdminTask.unconfigureLoginModule(['loginType', 'system', '-loginEntryAlias', 'systemLoginEntry', '-loginModule',
'moduleClass'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureLoginModule('-interactive')
```

createAuthDataEntry

The createAuthDataEntry command creates an authentication data entry for a J2EE Connector architecture (J2C) connector in the global security or security domain configuration.

Target object

None.

Required parameters

-alias
Specifies the name that uniquely identifies the authentication data entry. (String)

-user
Specifies the J2C authentication data user ID. (String)

-password
Specifies the password to use for the target enterprise information system (EIS). (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain configuration. The application server uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-description
Specifies a description of the authentication data entry. (String)

Return value

The command returns the object name of the new authentication data entry object.

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuthDataEntry('-alias dataEntry1 -user userID -password userIDpw')
```

- Using Jython list:

```
AdminTask.createAuthDataEntry(['alias', 'dataEntry1', '-user', 'userID', '-password', 'userIDpw'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuthDataEntry('-interactive')
```

deleteAuthDataEntry

The deleteAuthDataEntry command removes an authentication data entry for a J2C connector in a global security or security domain configuration.

Target object

None.

Required parameters

-alias
Specifies the name that uniquely identifies the authentication data entry. (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain configuration. The application server uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuthDataEntry('-alias dataEntry1')
```

- Using Jython list:

```
AdminTask.deleteAuthDataEntry(['alias', 'dataEntry1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuthDataEntry('-interactive')
```

getAuthDataEntry

The getAuthDataEntry command displays information about an authentication data entry for the J2C connector in the global security configuration or for a specific security domain.

Target object

None.

Required parameters

-alias
Specifies the name that uniquely identifies the authentication data entry. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an attribute list that contains the authentication data entry attributes and values.

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuthDataEntry('-alias authDataEntry1 -securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getAuthDataEntry(['alias', 'authDataEntry1', '-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuthDataEntry('-interactive')
```

listAuthDataEntries

The listAuthDataEntries command displays each authentication data entry in the global security configuration or in a security domain.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an array of attribute lists for each authentication data entry.

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuthDataEntries('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.listAuthDataEntries(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuthDataEntries('-interactive')
```

modifyAuthDataEntry

The modifyAuthDataEntry command modifies an authentication data entry for a J2C connector in the global security or security domain configuration.

Target object

None.

Required parameters

-alias
Specifies the name that uniquely identifies the authentication data entry. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-user
Specifies the J2C authentication data user ID. (String)

-password
Specifies the password to use for the target enterprise information system (EIS). (String)

-description
Specifies a description for the authentication data entry. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuthDataEntry('-alias dataEntry1 -user userID1 -password newPassword')
```

- Using Jython list:

```
AdminTask.modifyAuthDataEntry(['alias', 'dataEntry1', '-user', 'userID1', '-password', 'newPassword'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuthDataEntry('-interactive')
```

configureCSIInbound

The configureCSIInbound command configures CSIv2 inbound authentication on a security domain or on the global security configuration. When configuring CSI inbound authentication in a security domain for the first time that the CSI objects are copied from global security so thatany changes to that configuration are applied.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. If one is not provided the task will work on the global security user registry configuration. (String)

-messageLevelAuth
Specifies whether clients connecting to this server must specify a user ID and password. Specify `Never` to disable the user ID and password requirement. Specify `Supported` to accept a user ID and password. Specify `Required` to require a user ID and password. (String)

-supportedAuthMechList
Specifies the authentication mechanism to use. Specify `KRB5` for Kerberos authentication, `LTPA` for Lightweight Third-Party Authentication, `BasicAuth` for BasicAuth authentication, and `custom` to use your own authentication token implementation. You can specify more then one in a space-separated list. (String)

-clientCertAuth
Specifies whether a client that connects to the server must connect using an SSL certificate. Specify `Never` to allow clients to connect without SSL certificates. Specify `Supported` to accept clients connecting with and without SSL certificates. Specify `Required` to require clients to use SSL certificate. (String)

-transportLayer
Specifies the transport layer support level. Specify `Never` to disable transport layer support. Specify `Supported` to enable transport layer support. Specify `Required` to require transport layer support. (String)

-sslConfiguration
Specifies the SSL configuration alias to use for inbound transport. (String)

-enableIdentityAssertion
Specifies whether to enable identity assertion. When using the identity assertion authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element. Specify `true` for the -enableIdentityAssertion parameter to enable identity assertion. (Boolean)

-trustedIdentities
Specifies a list of trusted server identities, separated by the pipe character (|). To specify a null value, set the value of the -trustedIdentities parameter as an empty string (""). (String)

-statefulSession
Specifies whether to enable a stateful session. Specify `true` to enable a stateful session. (Boolean)

-enableAttributePropagation
Specifies whether to enable security attribute propagation. Security attribute propagation allows the application server to transport authenticated Subject contents and security context information from one server to another in your configuration. Specify `true` to enable security attribute propagation. (Boolean)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureCSIInbound('[-securityDomainName testDomain -messageLevelAuth Required
-supportedAuthMechList "KRB5 LTPA"]')
```

- Using Jython list:

```
AdminTask.configureCSIInbound(['-securityDomainName', 'testDomain', '-messageLevelAuth', 'Required',
'-supportedAuthMechList', 'KRB5 LTPA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureCSIInbound('-interactive')
```

configureCSIOutbound

The configureCSIOutbound command configures the CSIv2 outbound authentication in a security domain or in the global security configuration. When configuring CSI Outbound in a security domain for the first time, the application server copies the CSI objects from global security. Then, the application server applies the changes to that configuration from the command.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. (String)

-enableAttributePropagation
Specifies whether to enable security attribute propagation. Security attribute propagation allows the application server to transport authenticated Subject contents and security context information from one server to another in your configuration. Specify `true` to enable security attribute propagation. (Boolean)

-enableIdentityAssertion
Specifies whether to enable identity assertion. When using the identity assertion authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element. Specify `true` for the -enableIdentityAssertion parameter to enable identity assertion. (Boolean)

-useServerIdentity
Specifies whether to use the server identity to establish trust with the target server. Specify `true` to use the server identity. (Boolean)

-trustedId
Specifies the trusted identity that the application server uses to establish trust with the target server. (String)

-trustedIdentityPassword
Specifies the password of the trusted server identity. (String)

-messageLevelAuth
Specifies whether clients connecting to this server must specify a user ID and password. Specify `includeNever` to disable the user ID and password requirement. Specify `Supported` to accept a user ID and password. Specify `Required` to require a user ID and password. (String)

-supportedAuthMechList
Specifies the authentication mechanism to use. Specify `KRB5` for Kerberos authentication, `LTPA` for Lightweight Third-Party Authentication, `BasicAuth` for BasicAuth authentication, and `custom` to use your own authentication token implementation. You can specify more then one in a space-separated list. (String)

-clientCertAuth
Specifies whether a client that connects to the server must connect using an SSL certificate. Specify `Never` to allow clients to connect without SSL certificates. Specify `Supported` to accept clients connecting with and without SSL certificates. Specify `Required` to require clients to use SSL certificate. (String)

-transportLayer
Specifies the transport layer support level. Specify `Never` to disable transport layer support. Specify `Supported` to enable transport layer support. Specify `Required` to require transport layer support. (String)

-sslConfiguration
Specifies the SSL configuration alias to use for inbound transport. (String)

-statefulSession
Specifies whether to enable a stateful session. Specify `true` to enable a stateful session. (Boolean)

-enableOutboundMapping
Specifies whether to enable custom outbound identity mapping. Specify `true` to enable custom outbound identity mapping. (Boolean)

-trustedTargetRealms
Specifies a list of target realms to trust. Separate each realm name with the pipe character (|). (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureCSIOutbound('-securityDomainName testDomain -useServerIdentity true -messageAuthLevel
Supported')
```

- Using Jython list:

```
AdminTask.configureCSIOutbound(['securityDomainName', 'testDomain', '-useServerIdentity', 'true',
'-messageAuthLevel', 'Supported'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureCSIOutbound('-interactive')
```

getCSIInboundInfo

The getCSIInboundInfo command displays information about the Common Secure Interoperability (CSI) inbound settings for the global security configuration or for a security domain.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-displayModel
Specifies the output format of the configuration information. Specify `true` to return an attribute list of the model. Specify `false` to display an attribute of the value used to create the object. (Boolean)

Return value

The command returns an attribute list of the attributes and values of the CSI inbound object.

Batch mode example usage

- Using Jython string:

```
AdminTask.getCSIInboundInfo('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getCSIInboundInfo(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getCSIInboundInfo('-interactive')
```

getCSIOutboundInfo

The getCSIOutboundInfo command displays information for the CSI outbound settings for the global security configuration or for a security domain.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

-displayModel
Specifies the output format of the configuration information. Specify `true` to return an attribute list of the model. Specify `false` to display an attribute of the value used to create the object. (Boolean)

Return value

The command returns an attribute list that contains the attributes and values of the CSI outbound configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.getCSIOutboundInfo('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getCSIOutboundInfo(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getCSIOutboundInfo('-interactive')
```

unconfigureCSIInbound

The unconfigureCSIInbound command removes the CSI inbound information from a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureCSIInbound('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureCSIInbound(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureCSIInbound('-interactive')
```

unconfigureCSIOutbound

The unconfigureCSIOutbound command removes the CSI outbound information from a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureCSIOutbound('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureCSIOutbound(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureCSIOutbound('-interactive')
```

configureInterceptor

The configureInterceptor command modifies an existing interceptor or creates an interceptor if one does not exist.

Target object

None.

Required parameters

-interceptor
Specifies the trust association interceptor class name. (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain. If you do not specify a security domain, the command assigns the global security configuration. (String)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: ″attr1=value1″,″attr2=value2″ (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureInterceptor('-interceptor com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus
-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.configureInterceptor(['interceptor', 'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus',
'-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureInterceptor('-interactive')
```

configureTrustAssociation

The configureTrustAssociation command enables or disable the trust association. If the security domain does not have a trust association defined, the application server copies each trust association and its interceptors from the global security configuration.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. (String)

-enable
Specifies whether to enable trust association to act as a reverse proxy server. (Boolean)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureTrustAssociation('-securityDomainName testDomain -enable true')
```

- Using Jython list:

```
AdminTask.configureTrustAssociation(['securityDomainName', 'testDomain', '-enable', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureTrustAssociation('-interactive')
```

getTrustAssociationInfo

The getTrustAssociationInfo command displays configuration information for trust association.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an attribute list that contains attributes and values for trust association.

Batch mode example usage

- Using Jython string:

```
AdminTask.getTrustAssociationInfo('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getTrustAssociationInfo(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getTrustAssociationInfo('-interactive')
```

listInterceptors

The listInterceptors command displays the trust association interceptors that are configured in the global security or security domain configuration.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an array list of each interceptor and the associated custom properties.

Batch mode example usage

- Using Jython string:

```
AdminTask.listInterceptors('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.listInterceptors(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listInterceptors('-interactive')
```

unconfigureInterceptor

The unconfigureInterceptor command removes a trust association interceptor from the global security configuration or from a security domain.

Target object

None.

Required parameters

-interceptor
Specifies the trust association interceptor class name. (String)

Optional parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureInterceptor('-interceptor com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus
-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureInterceptor(['interceptor', 'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus',
'-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureInterceptor('-interactive')
```

unconfigureTrustAssociation

The unconfigureTrustAssociation command removes the trust association object from a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureTrustAssociation('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureTrustAssociation(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureTrustAssociation('
```

configureAuthzConfig

The configureAuthzConfig command configures an external Java Authorization Contract for Containers (JACC) authorization provider in a security domain or the global security configuration.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security configuration. (String)

-useJACCProvider
Specifies whether to use a JACC provider. Specify `true` to use a JACC provider. (Boolean)

-name
Specifies the name of the JACC provider to use. (String)

-description
Specifies a description of the JACC provider. (String)

-j2eePolicyImplClassName
Specifies the class name of an implementation class that represents the javax.security.jacc.policy.provider property according to the specification. (String)

-policyConfigurationFactoryImplClassName
Specifies the class name of an implementation class that represents the javax.security.jacc.PolicyConfigurationFactory.provider property. (String)

-roleConfigurationFactoryImplClassName
Specifies the class name of an implementation class that implements the com.ibm.wsspi.security.authorization.RoleConfigurationFactory interface. (String)

-requiresEJBArgumentsPolicyContextHandler
Specifies whether policy providers require the Enterprise JavaBeans arguments policy context handler to make access decisions. Specify `true` to enable this option. (Boolean)

-initializeJACCProviderClassName
Specifies the class name of an implementation class that implements the com.ibm.wsspi.security.authorization.IntializeJACCProvider interface.(String)

-supportsDynamicModuleUpdates
Specifies whether the provider supports dynamic changes to the Web modules. Specify `true` to enable this option. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureAuthzConfig('[-securityDomainName testDomain -useJACCProvider true -name
testProvider -description "JACC provider for testing"]')
```

- Using Jython list:

```
AdminTask.configureAuthzConfig(['securityDomainName', 'testDomain', '-useJACCProvider', 'true', '-name',
'testProvider', '-description', 'JACC provider for testing'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureAuthzConfig('-interactive')
```

configureSingleSignon

The configureSingleSignon command configures a single sign-on object in global security.

Target object

None.

Optional parameters

-enable
Specifies whether to enable single sign-on. Specify `true` to enable single sign-on, or `false` to disable single sign-on. (Boolean)

-requiresSSL
Specifies whether single sign-on requests send through HTTPS. Specify `true` to enable this option. (Boolean)

-domainName
Specifies the domain name that contains a set of hosts to which the single sign-on applies. (String)

-interoperable
Specifies interoperability options. Specify `true` to send an interoperable cookie to the browser to support back-level servers. Specify `false` disable the sending of interoperable cookies. (Boolean)

-attributePropagation
Specifies whether to enable inbound security attribute propagation. Specify `true` to enable Web inbound security attribution propagation. Specify `false` to use the single sign-on token to log in and recreate the Subject from the user registry. (Boolean)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureSingleSignon('-enable true -domainName mycompany.com')
```

- Using Jython list:

```
AdminTask.configureSingleSignon(['enable', 'true', '-domainName', 'mycompany.com'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.configureSingleSignon('-interactive')
```

getActiveSecuritySettings

The getActiveSecuritySettings command displays the active security settings for global security or a specific security domain.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security domain configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns the active security settings for the security domain of interest or the global security configuration, which includes the following settings:

- cacheTimeout
- issuePermissionWarning
- activeAuthMechanism
- enforceJava2Security
- appSecurityEnabled
- enableGlobalSecurity (global security only)
- adminPreferredAuthMech (global security only)
- activeAuthMechanism (global security only)
- activeUserRegistry
- enforceFineGrainedJCASecurity
- dynUpdateSSLConfig (global security only)
- useDomainQualifiedUserNames
- customProperties

Batch mode example usage

- Using Jython string:

```
AdminTask.getActiveSecuritySettings('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getActiveSecuritySettings(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getActiveSecuritySettings('-interactive')
```

getAuthzConfigInfo

The getAuthzConfigInfo command displays information about an external JACC authorization provider in a security domain or the global security configuration.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security domain configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command returns an attribute list that contains the attributes and values that are associated with the JACC authorization provider.

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuthzConfigInfo('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.getAuthzConfigInfo(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuthzConfigInfo('-interactive')
```

getSingleSignon

The getSingleSignon command displays configuration information about the single sign-on object as defined in the global security configuration.

Target object

None.

Optional parameters

None.

Return value

The command returns an attribute list that contains the attributes and values of the single sign-on configuration.

Batch mode example usage

- Using Jython:

```
AdminTask.getSingleSignon()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getSingleSignon('-interactive')
```

setAdminActiveSecuritySettings

The setAdminActiveSecuritySettings command sets the active security settings on the global security object.

Target object

None.

Optional parameters

-enableGlobalSecurity
Specifies whether to enable global security. Specify `true` to enable global security, or specify `false` to disable global security. (Boolean)

-cacheTimeout
Specifies the amount of time, in seconds, before authentication data becomes invalid. (Integer)

-issuePermissionWarning
Specifies whether to issue a warning during application installation if the application requires security permissions. Specify `true` to enable the warning notification, or specify `false` to disable the warning notification. (Boolean)

-enforceJava2Security
Specifies whether to enable Java Platform, Enterprise Edition (Java EE) security. Specify `true` to enable Java EE security permissions checking, or specify `false` to disable Java EE security. (Boolean)

-enforceFineGrainedJCASecurity
Specifies whether to restrict application access. Specify `true` to restrict application access to sensitive Java EE Connector Architecture (JCA) mapping authentication data. (Boolean)

-appSecurityEnabled
Specifies whether to enable application-level security. Specify `true` to enable application level security, or specify `false` to disable application-level security. (Boolean)

-dynUpdateSSLConfig
Specifies whether to dynamically update SSL configuration changes. Specify `true` to update SSL configuration changes dynamically, or specify `false` to update the SSL configuration when the server starts. (Boolean)

-activeAuthMechanism
Specifies the active authentication mechanism. Specify `LTPA` for LTPA authentication, `KRB5` for Kerberos authentication, or `RSAToken` for RSA token authorization. (String)

-adminPreferredAuthMech
Specifies the preferred authentication mechanism. Specify `LTPA` for LTPA authentication, `KRB5` for Kerberos authentication, or `RSAToken` for RSA token authorization. (String)

-activeUserRegistry
Specifies the active user registry for the server. (String)

-useDomainQualifiedUserNames
Specifies the type of user name to use. Specify `true` to use domain qualified user names, or specify `false` to use the short name. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAdminActiveSecuritySettings('-enableGlobalSecurity true -cacheTimeout 300
-enforceJava2Security true -appSecurityEnabled true')
```

- Using Jython list:

```
AdminTask.setAdminActiveSecuritySettings(['enableGlobalSecurity', 'true', '-cacheTimeout',
'300', '-enforceJava2Security', 'true', '-appSecurityEnabled', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setAdminActiveSecuritySettings('-interactive')
```

setAppActiveSecuritySettings

The setAppActiveSecuritySettings command sets the active security settings on a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Optional parameters

-cacheTimeout
Specifies the amount of time, in seconds, before authentication data becomes invalid. (Integer)

-issuePermissionWarning
Specifies whether to issue a warning during application installation if the application requires security permissions. Specify `true` to enable the warning notification, or specify `false` to disable the warning notification. (Boolean)

-enforceJava2Security
Specifies whether to enable Java Platform, Enterprise Edition (Java EE) security. Specify `true` to enable Java EE security permissions checking, or specify `false` to disable Java EE security. (Boolean)

-enforceFineGrainedJCASecurity
Specifies whether to restrict application access. Specify `true` to restrict application access to sensitive Java EE Connector Architecture (JCA) mapping authentication data. (Boolean)

-appSecurityEnabled
Specifies whether to enable application-level security. Specify `true` to enable application level security, or specify `false` to disable application-level security. (Boolean)

-activeUserRegistry
Specifies the active user registry for the server. (String)

-useDomainQualifiedUserNames
Specifies the type of user name to use. Specify `true` to use domain qualified user names, or specify `false` to use the short name. (Boolean)

-customProperties
Specifies a comma separated list of quoted attribute and value pairs that the system stores as custom properties on the user registry object. For example, use the format: `"attr1=value1","attr2=value2"` (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAppActiveSecuritySettings('-securityDomainName testDomain -issuePermissionWarning false
-enforceFineGrainedJCASecurity true')
```

- Using Jython list:

```
AdminTask.setAppActiveSecuritySettings(['securityDomainName', 'testDomain', '-issuePermissionWarning',
'false', '-enforceFineGrainedJCASecurity', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setAppActiveSecuritySettings('-interactive')
```

unconfigureAuthzConfig

The unconfigureAuthzConfig command removes an external JACC authorization provider from the global security configuration or a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureAuthzConfig('-securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureAuthzConfig(['securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unconfigureAuthzConfig('-interactive')
```

unsetAppActiveSecuritySettings

The unsetAppActiveSecuritySettings command removes an attribute from the global security configuration or a security domain.

Target object

None.

Required parameters

-securityDomainName
Specifies the name of the security configuration. The command uses the global security configuration if you do not specify a value for the -securityDomainName parameter. (String)

Optional parameters

-unsetAppSecurityEnabled
Specifies whether to remove the attribute that enables application security. Specify `true` to remove the attribute. (Boolean)

-unsetActiveUserRegistry
Specifies whether to remove the active user registry attribute. Specify `true` to remove the attribute. (Boolean)

-unsetUseDomainQualifiedUserNames
Specifies whether to remove the user domain qualified user names attribute. Specify `true` to remove the attribute. (Boolean)

-unsetEnforceJava2Security
Specifies whether to remove the Java EE security attribute. Specify `true` to remove the attribute. (Boolean)

-unsetEnforceFineGrainedJCASecurity
Specifies whether to remove the fine-grained JCA security attribute. Specify `true` to remove the attribute. (Boolean)

-unsetIssuePermissionWarning
Specifies whether to remove the attribute that issues user permission warnings. Specify `true` to remove the attribute. (Boolean)

-unsetCacheTimeout
Specifies whether to remove the cache timeout attribute. Specify `true` to remove the attribute. (Boolean)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unsetAppActiveSecuritySettings('-securityDomainName testDomain -unsetAppSecurityEnabled true
-unsetPermissionWarning true')
```

- Using Jython list:

```
AdminTask.unsetAppActiveSecuritySettings(['securityDomainName', 'testDomain', '-unsetAppSecurityEnabled',
'true', '-unsetPermissionWarning', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unsetAppActiveSecuritySettings('-interactive')
```

SecurityRealmInfoCommands command group for the AdminTask object

You can use the Jython scripting language to manage security realm configurations with the wsadmin tool. Use the commands and parameters in the SecurityRealmInfoCommands group to query and manage trusted realms.

Use the following commands to manage trusted realms in your security configuration:

- “addTrustedRealms” on page 613
- “configureTrustedRealms” on page 613
- “listRegistryGroups” on page 614
- “listRegistryUsers” on page 615
- “listSecurityRealms” on page 616

addTrustedRealms

The addTrustedRealms command adds a realm or list of realms to the list of trusted realms for global security or in a security domain.

Target object

None.

Required parameters

-communicationType
Specifies whether to trusted realms to inbound or outbound communication. Specify `inbound` to configure inbound communication. Specify `outbound` to configure outbound communication. (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain of interest. If you do not specify a value for this parameter, the command uses the global security configuration. (String)

-realmList
Specifies a realm or list of realms to configure as trusted realms. (String)

Separate each realm in the list with the pipe character (|) as the following example demonstrates: `realm1|realm2|realm3`

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.addTrustedRealms('-communicationType inbound -securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.addTrustedRealms(['-communicationType', 'inbound', '-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addTrustedRealms('-interactive')
```

configureTrustedRealms

The configureTrustedRealms command configures trusted realms. Use this command to replace the list of trusted realms and to clear each realm from the list. To add realms to the trusted realm list, use the addInboundTrustedRealm command.

Target object

None.

Required parameters

-communicationType
Specifies whether to configure the security domains, realms, or global security configuration for inbound or outbound communication. Specify `inbound` to configure inbound communication. Specify `outbound` to configure outbound communication. (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain of interest. If you do not specify a value for this parameter, the command uses the global security configuration. (String)

-realmList
Specifies a list of realms to configure as trusted realms. (String)

Separate each realm in the list with the pipe character (|) as the following example demonstrates: `realm1|realm2|realm3`

-trustAllRealms
Specifies whether to trust all realms. Specify `true` to trust all realms. If you specify `true` for this parameter, the command does not use the -realmList parameter. (Boolean)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.configureTrustedRealms('-communicationType inbound -realmList realm1|realm2|realm3')
```

- Using Jython list:

```
AdminTask.configureTrustedRealms(['-communicationType', 'inbound', '-realmList', 'realm1|realm2|realm3'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.configureTrustedRealms('-interactive')
```

- Using Jython list:

listRegistryGroups

The listRegistryGroups command displays the groups in the user registry that belong to the security realm, security domain, or resource name of interest.

Target object

None.

Optional parameters

-securityRealmName
Specifies name of the security realm of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these parameters. (String)

-resourceName
Specifies the name of the resource of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these parameters. (String)

-securityDomainName
Specifies the name of the security domain of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these parameters.(String)

-displayAccessIds
Specifies whether to display the access IDs for each group. Specify `true` to display the access ID and group name for each group that the command returns. (Boolean)

-groupFilter
Specifies a filter that the command uses to query for groups. For example, specify `test*` to return groups that begin with the `test` string. By default, the command returns all groups. (String)

-numberOfGroups
Specifies the number of groups to return. The default number of groups that the command displays is `20`. (Integer)

Return value

The command returns an array of group names. If you specified the -displayAccessId parameter, the command returns an array of attribute lists which contain the group name and group access ID.

Batch mode example usage

- Using Jython string:

```
AdminTask.listRegistryGroups('-securityDomainName myTestDomain -groupFilter test* -numberOfGroups 10')
```

- Using Jython list:

```
AdminTask.listRegistryGroups(['-securityDomainName', 'myTestDomain', '-groupFilter', 'test*', '-numberOfGroups', '10'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listRegistryGroups('-interactive')
```

listRegistryUsers

The listRegistryUsers command displays the users in the user registry for a specific security realm, resource name, or domain name.

Target object

None.

Optional parameters

-securityDomainName
Specifies the name of the security domain of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these parameters. If you do not specify the securityDomainName, resourceName, or securityRealmName parameter, the system uses the active user registry from the global security configuration. (String)

-resourceName
Specifies the name of the resource of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these parameters. If you do not specify the securityDomainName, resourceName, or securityRealmName parameter, the system uses the active user registry from the global security configuration. (String)

-securityRealmName
Specifies the name of the security realm of interest. The securityDomainName, resourceName, and securityRealmName parameters are mutually exclusive. Do not specify more than one of these

parameters. If you do not specify the securityDomainName, resourceName, or securityRealmName parameter, the system uses the active user registry from the global security configuration. (String)

-displayAccessIds
Specifies whether to display the access IDs for each group. Specify `true` to display the access ID and group name for each group that the command returns. (Boolean)

-userFilter
Specifies the filter that the command uses to query for users. For example, specify `test*` to display each user name that starts with the `test` string. By default, the command returns all users. (String)

-numberOfUsers
Specifies the number of users to return. The default number of groups that the command displays is `20`. (Integer)

Return value

The command returns an array of user names. If you specify the -displayAccessId parameter, the command returns an array of attribute lists that contain the user ID and user access IDs.

Batch mode example usage

- Using Jython string:

```
AdminTask.listRegistryUsers('-securityRealmName defaultWIMFileBasedRealm  -displayAccessIds true')
```

- Using Jython list:

```
AdminTask.listRegistryUsers(['-securityRealmName', 'defaultWIMFileBasedRealm', '-displayAccessIds', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listRegistryUsers('-interactive')
```

listSecurityRealms

The listSecurityRealms command displays each security realm from global security configuration and the security domains.

Target object

None.

Return value

The command returns an array of realm names.

Batch mode example usage

- Using Jython string:

```
AdminTask.listSecurityRealms()
```

- Using Jython list:

```
AdminTask.listSecurityRealms()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSecurityRealms('-interactive')
```

listTrustedRealms

The listTrustedRealms command displays a list of trusted realms for a security domain, resource, or realm. If you do not specify a security domain, resource name, or realm name, then the command returns a list of trusted realms from the global security configuration. The securityRealmName, resourceName, and securityDomainName parameters are mutually exclusive.

Target object

None.

Required parameters

-communicationType
Specifies whether to list the trusted realms for inbound or outbound communication. Specify `inbound` to configure inbound communication. Specify `outbound` to configure outbound communication. (String)

Optional parameters

-securityRealmName
Specifies name of the security realm of interest. If you use this parameter, do not use the resourceName or securityDomainName parameters. (String)

-resourceName
Specifies the name of the resource of interest. If you use this parameter, do not use the securityRealmName or securityDomainName parameters. (String)

-securityDomainName
Specifies the name of the security domain of interest. If you use this parameter, do not use the resourceName or securityRealmName parameters. (String)

-expandRealmList
Specifies whether to return each realm name when the trustAllRealms property is enabled. Specify `true` to return each realm name. Specify `false` to return the trustAllRealms property. (Boolean)

-includeCurrentRealm
Specifies whether to include the current realm in the list of trusted realms. Specify `true` to include the current realm, or specify `false` to exclude the current realm from the list of trusted realms. (Boolean)

Return value

The command returns an array of trusted realm names. If the realm, resource, or security domain of interest is configured to trust all realms, the command returns the `trustAllRealms` string.

Batch mode example usage

- Using Jython string:

```
AdminTask.listTrustedRealms('-communicationType inbound -resourceName myApplication')
```

- Using Jython list:

```
AdminTask.listTrustedRealms(['-communicationType', 'inbound', '-resourceName', 'myApplication'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listTrustedRealms('-interactive')
```

removeTrustedRealms

The removeTrustedRealms command removes realms from a trusted realm list in a security domain or in the global security configuration.

Target object

None.

Required parameters

-communicationType
Specifies whether to remove trusted realms from inbound or outbound communication. Specify `inbound` to configure inbound communication. Specify `outbound` to configure outbound communication. (String)

-realmList
Specifies a list of realms to remove from trusted realms. (String)

Separate each realm in the list with the pipe character (|) as the following example demonstrates: `realm1|realm2|realm3`

Optional parameters

-securityDomainName
Specifies the name of the security domain of interest. If you do not specify a security domain, the command uses the global security configuration. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.removeTrustedRealms('-communicationType inbound -realmList realm1|realm2|realm3')
```

- Using Jython list:

```
AdminTask.removeTrustedRealms(['-communicationType inbound -realmList realm1|realm2|realm3'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeTrustedRealms('-interactive')
```

unconfigureTrustedRealms

The unconfigureTrustedRealms command removes the trusted realm object from the configuration.

Target object

None.

Required parameters

-communicationType
Specifies whether to unconfigure the trusted realms for inbound or outbound communication. Specify `inbound` to remove inbound communication configurations. Specify `outbound` to remove outbound communication configurations. (String)

Optional parameters

-securityDomainName
Specifies the name of the security domain of interest. If you do not specify a security domain, the command uses the global security configuration. (String)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.unconfigureTrustedRealms('-communicationType inbound -securityDomainName testDomain')
```

- Using Jython list:

```
AdminTask.unconfigureTrustedRealms(['-communicationType', 'inbound', '-securityDomainName', 'testDomain'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.unconfigureTrustedRealms('-interactive')
```

NamingAuthzCommands command group for the AdminTask object

You can use the Jython scripting language to configure naming roles for groups and users with the wsadmin tool. Use the commands and parameters in the NamingAuthzCommands group to assign, remove, and query naming role configuration. CosNaming security offers increased granularity of security control over CosNaming functions.

A number of naming roles are defined to provide the degrees of authority that are needed to perform certain application server naming service functions. The authorization policy is only enforced when global security is enabled.

Use the following commands to manage the naming service functions:

- listGroupsForNamingRoles
- listUsersForNamingRoles
- mapGroupsToNamingRole
- mapUsersToNamingRole
- removeGroupsFromNamingRole
- removeUsersFromNamingRole

listGroupsForNamingRoles

The listGroupsForNamingRoles command displays the groups and special subjects that are mapped to the naming roles.

Target object

None.

Return value

The command returns a list of the groups and special subjects associated with each naming role.

Batch mode example usage

- Using Jython:

```
AdminTask.listGroupsForNamingRoles()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listGroupsForNamingRoles('-interactive')
```

listUsersForNamingRoles

The listUsersForNamingRoles command displays the users that are mapped to the naming roles.

Target object

None.

Return value

The command returns a list of the users associated with each naming role.

Batch mode example usage

- Using Jython:

```
AdminTask.listUsersForNamingRoles()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listUsersForNamingRoles('-interactive')
```

mapGroupsToNamingRole

The mapGroupsToNamingRole command maps groups, special subjects, or groups and special subjects to the naming roles.

Target object

None.

Required parameters

-roleName
Specifies the name of the naming role. (String)

Four name space security roles are available: `CosNamingRead`, `CosNamingWrite`, `CosNamingCreate`, and `CosNamingDelete`. The roles have authority levels from low to high, as the following table defines:

Role name	Description
`CosNamingRead`	You can query the application server name space using, for example, the Java Naming and Directory Interface (JNDI) lookup method. The `EVERYONE` special-subject is the default policy for this role.
`CosNamingWrite`	You can perform write operations such as JNDI bind, rebind, or unbind, and CosNamingRead operations.
`CosNamingCreate`	You can create new objects in the name space through operations such as JNDI createSubcontext and CosNamingWrite operations.
`CosNamingDelete`	You can destroy objects in the name space, for example using the JNDI destroySubcontext method and CosNamingCreate operations.

Optional parameters

-groupids
Specifies the names of the groups to map to the naming roles. (String[])

-accessids
Specifies the access IDs of the users in the format `<group:realmName/uniqueID>`. (String[])

-specialSubjects
Specifies the special subjects to map. (String[])

The special subjects include EVERYONE, ALLAUTHENTICATED, ALLAUTHENTICATEDINTRUSTEDREALMS, as the following table defines:

Header	Header
EVERYONE	Maps everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.
ALLAUTHENTICATED	Maps each authenticated user to a specified role. When you map each authenticated user to a specified role, each valid user in the current registry who has been authenticated can access resources that are protected by this role.
ALLAUTHENTICATEDINTRUSTEDREALMS	Maps each authenticated user to a specified role. When you map each authenticated user to a specified role, each valid user in the current registry who has been authenticated can access resources that are protected by this role in the trusted realm.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.mapGroupsToNamingRole('[-roleName CosNamingCreate -groupids [group1, group2]]')
```

- Using Jython list:

```
AdminTask.mapGroupsToNamingRole(['-roleName', 'CosNamingCreate', '-groupids', '[group1, group2]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.mapGroupsToNamingRole('-interactive')
```

mapUsersToNamingRole

The mapUsersToNamingRole command maps users to the naming roles.

Target object

None.

Required parameters

-roleName
Specifies the name of the naming role. (String)

Four name space security roles are available: CosNamingRead, CosNamingWrite, CosNamingCreate, and CosNamingDelete. The roles have authority levels from low to high, as the following table defines:

Role name	Description
CosNamingRead	You can query the application server name space using, for example, the Java Naming and Directory Interface (JNDI) lookup method. The EVERYONE special-subject is the default policy for this role.
CosNamingWrite	You can perform write operations such as JNDI bind, rebind, or unbind, and CosNamingRead operations.
CosNamingCreate	You can create new objects in the name space through operations such as JNDI createSubcontext and CosNamingWrite operations.
CosNamingDelete	You can destroy objects in the name space, for example using the JNDI destroySubcontext method and CosNamingCreate operations.

Optional parameters

-userids
Specifies the user IDs to map to the naming roles of interest. (String[])

-accessids
Specifies the access IDs of the users in the format <user:realmName/uniqueID>. (String[])

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.mapUsersToNamingRole([-roleName CosNamingDelete -userids [user1, user2, user3]]')
```

- Using Jython list:

```
AdminTask.mapUsersToNamingRole(['-roleName', 'CosNamingDelete', '-userids', '[user1, user2, user3]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.mapUsersToNamingRole('-interactive']
```

removeGroupsFromNamingRole

The removeGroupsFromNamingRole command removes groups, special subjects, or groups and special subjects from a naming role.

Target object

None.

Required parameters

-roleName
Specifies the name of the naming role. (String)

Four name space security roles are available: CosNamingRead, CosNamingWrite, CosNamingCreate, and CosNamingDelete. The roles have authority levels from low to high, as the following table defines:

Role name	Description
CosNamingRead	You can query the application server name space using, for example, the Java Naming and Directory Interface (JNDI) lookup method. The EVERYONE special-subject is the default policy for this role.
CosNamingWrite	You can perform write operations such as JNDI bind, rebind, or unbind, and CosNamingRead operations.
CosNamingCreate	You can create new objects in the name space through operations such as JNDI createSubcontext and CosNamingWrite operations.
CosNamingDelete	You can destroy objects in the name space, for example using the JNDI destroySubcontext method and CosNamingCreate operations.

Optional parameters

-groupids
Specifies the names of the groups to remove from the naming roles of interest. (String[])

-specialSubjects
Specifies the special subjects to remove. (String[])

The special subjects include EVERYONE, ALLAUTHENTICATED, ALLAUTHENTICATEDINTRUSTEDREALMS, as the following table defines:

Header	Header
EVERYONE	Maps everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.

Header	Header
ALLAUTHENTICATED	Maps each authenticated user to a specified role. When you map each authenticated user to a specified role, each valid user in the current registry who has been authenticated can access resources that are protected by this role.
ALLAUTHENTICATEDINTRUSTEDREALMS	Maps each authenticated user to a specified role. When you map each authenticated user to a specified role, each valid user in the current registry who has been authenticated can access resources that are protected by this role in the trusted realm.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.removeGroupsFromNamingRole('-roleName CosNamingRead -groupids [group1, group2] -specialSubjects EVERYONE')
```

- Using Jython list:

```
AdminTask.removeGroupsFromNamingRole(['-roleName', 'CosNamingRead', '-groupids', '[group1, group2]', '-specialSubjects', 'EVERYONE'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeGroupsFromNamingRole('-interactive')
```

removeUsersFromNamingRole

The removeUsersFromNamingRole command removes users from a naming role.

Target object

None.

Required parameters

-roleName
Specifies the name of the naming role. (String)

Four name space security roles are available: CosNamingRead, CosNamingWrite, CosNamingCreate, and CosNamingDelete. The roles have authority levels from low to high, as the following table defines:

Role name	Description
CosNamingRead	You can query the application server name space using, for example, the Java Naming and Directory Interface (JNDI) lookup method. The EVERYONE special-subject is the default policy for this role.
CosNamingWrite	You can perform write operations such as JNDI bind, rebind, or unbind, and CosNamingRead operations.
CosNamingCreate	You can create new objects in the name space through operations such as JNDI createSubcontext and CosNamingWrite operations.
CosNamingDelete	You can destroy objects in the name space, for example using the JNDI destroySubcontext method and CosNamingCreate operations.

Optional parameters

-userids
Specifies the user IDs to remove from the naming roles of interest. (String[])

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.removeUsersFromNamingRole('-roleName CosNamingRead')
```

- Using Jython list:

```
AdminTask.removeUsersFromNamingRole(['-roleName', 'CosNamingRead'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeUsersFromNamingRole('-interactive')
```

Utility scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that set notification options, save configuration changes, and display scripting library information.

Each utility script procedure is located in the *app_server_root*/scriptLibraries/utilities/V70 directory. Use the following script procedures to perform utility functions:

convertToList

This script converts a string to a list. For example, the `AdminApp.list()` command returns a string of application names. Use the convertToList script to change the output to a list format, such as `['DefaultApplication', 'a1', 'a2', 'ivtApp', 'query']`.

To run the script, run a command that returns a string output and set the output to a variable, as defined in the following table:

Argument	Description
variable	Specifies the name of the variable that contains the string to convert to a list.

Syntax

```
AdminUtilities.convertToList(variable)
```

Example usage

```
apps=AdminApp.list()
AdminUtilities.convertToList(apps)
```

debugNotice

This script sets the debug notice text.

To run the script, specify the message argument, as defined in the following table:

Argument	Description
message	Specifies the message text for the debug notice.

Syntax

```
AdminUtilities.debugNotice(message)
```

Example usage

```
AdminUtilities.debugNotice("Server is started")
```

getExceptionText

This script displays the exception message for a specific exception type, exception value, or traceback information.

To run the script, specify the type, value, or traceback arguments, as defined in the following table:

Argument	Description
type	Specifies the exception type of interest. The exception type represents the class object of the exception.
value	Specifies the exception value of interest. The value represents the instance object that is the argument of the exception or the second argument of the raise statement.
traceback	Specifies the traceback information of interest. The traceback object contains special attributes, including the line number where the error occurred. Do not assign *traceback* to a local variable in the function that handles the exception, as this assignment creates a circular reference.

Syntax

```
AdminUtilities.getExceptionText(type, value, traceback)
```

Example usage

```
AdminUtilities.getExceptionText("com.ibm.ws.scripting.ScriptingException
", "com.ibm.ws.scripting.ScriptingException: AdminControl service not available"
, "")
```

fail

This script sets the failure message.

To run the script, specify the message argument, as defined in the following table:

Argument	Description
message	Specifies the message text for the failure notice.

Syntax

```
AdminUtilities.fail(message)
```

Example usage

```
AdminUtilities.fail("The script failed")
```

fileSearch

This script searches the file system based on a specific path or directory.

To run the script, specify the path or directory arguments, as defined in the following table:

Argument	Description
path	Specifies the file path to search for a specific file.
directory	Specifies the directory to search for a specific file.

Syntax

```
AdminUtilities.fileSearch(path, directory)
```

Example usage

```
Paths = []
Directory = java.io.File("//WebSphere//AppServer//scriptLibraries")
AdminUtilities.fileSearch(directory, paths)
```

getResourceBundle

This script displays an instance for the resource bundle of interest.

To run the script, specify the bundle name argument, as defined in the following table:

Argument	Description
bundleName	Specifies the name of the bundle of interest. For example, to get a message object from the ScriptingLibraryMessage resource bundle, specify `com.ibm.ws.scripting.resources.scriptLibraryMessage`.

Syntax

```
AdminUtilities.getResourceBundle(bundleName)
```

Example usage

```
AdminUtilities.getResourceBundle("com.ibm.ws.scripting.resources.scriptLibraryMessage")
```

getScriptLibraryFiles

This script displays the file path and file names for each script library file.

Syntax

```
AdminUtilities.getScriptLibraryFiles()
```

Example usage

```
AdminUtilities.getScriptLibraryFiles()
```

getScriptLibraryList

This script displays each script name in the script library.

Syntax

```
AdminUtilities.getScriptLibraryList()
```

Example usage

```
AdminUtilities.getScriptLibraryList()
```

getScriptLibraryPath

This script displays the file path to get to the script library files on your file system.

Syntax

```
AdminUtilities.getScriptLibraryPath()
```

Example usage

```
AdminUtilities.getScriptLibraryPath()
```

help

This script displays help information for the AdminUtilities script library, including general library information, script names, and script descriptions.

To run the script, optionally specify the name of the script of interest, as defined in the following table:

Argument	Description
scriptName	Optionally specifies the name of the AdminUtilities script of interest.

Syntax

```
AdminUtilities.help(scriptName)
```

Example usage

```
AdminUtilities.help("sleepDelay")
```

infoNotice

This script sets the text for the information notice of a command or script.

To run the script, specify the message argument, as defined in the following table:

Argument	Description
message	Specifies the message text or a message ID such as "Application is installed" or resourceBundle.getString("WASX7115I").

Syntax

```
AdminUtilities.infoNotice(message)
```

Example usage

```
AdminUtilities.infoNotice(resourceBundle.getString("WASX7115I"))
```

save

This script saves the configuration changes to your system.

Syntax

```
AdminUtilities.save()
```

Example usage

```
AdminUtilities.save()
```

setDebugNotices

This script enables and disables debug notices.

To run the script, specify the debug argument, as defined in the following table:

Argument	Description
debug	Specifies whether to enable or disable debug notices. Specify `true` to enable debug notices, or `false` to disable debug notices.

Syntax

```
AdminUtilities.setDebugNotices(debug)
```

Example usage

```
AdminUtilities.setDebugNotices("true")
```

setFailOnErrorDefault

This script enables or disables the fail on error behavior.

To run the script, specify the fail on error argument, as defined in the following table:

Argument	Description
failOnError	Specifies whether to enable or disable the fail on error behavior. Specify `true` to enable the fail on error behavior, or `false` to disable the behavior.

Syntax

```
AdminUtilities.setFailOnErrorDefault(failOnError)
```

Example usage

```
AdminUtilities.setFailOnErrorDefault("false")
```

sleepDelay

This script sets the number of seconds that the system waits for completion during two operations.

To run the script, specify the delay seconds argument, as defined in the following table:

Argument	Description
delaySeconds	Specifies the number of seconds to wait for completion.

Syntax

```
AdminUtilities.sleepDelay(delaySeconds)
```

Example usage

```
AdminUtilities.sleepDelay("10")
```

warningNotice

This script sets the text to display as the warning message.

To run the script, specify the message argument, as defined in the following table:

Argument	Description
message	Specifies the non-translated text for the warning notice or a message ID such as `resourceBundle.getString("WASX7411W")`.

Syntax

```
AdminUtilities.warningNotice(message)
```

Example usage

```
AdminUtilities.warningNotice(resourceBundle.getString("WASX7411W"))
```

Configuring the JACC provider for Tivoli Access Manager using the wsadmin utility

You can use the wsadmin utility to configure Tivoli Access Manager security for WebSphere Application Server.

About this task

Verify that all the managed servers, including node agents, are started. The following configuration is performed once on the deployment manager server. The configuration parameters are forwarded to managed servers, including node agents, when a synchronization is performed. The managed servers require their own restart for the configuration changes to take effect.

1. Start WebSphere Application Server.
2. At the **wsadmin** prompt, enter the following command:

   $AdminTask configureTAM -interactive

   You are prompted to enter the following information:

Option	Description
WebSphere Application Server node name	Specify a single node or enter an asterisk (*) to choose all nodes including the deployment manager.
Tivoli Access Manager Policy Server	Enter the name of the Tivoli Access Manager policy server and the connection port. Use the format, *policy_server* **:** *port*. The policy server communication port is set at the time of Tivoli Access Manager configuration. The default port is 7135.
Tivoli Access Manager Authorization Server	Enter the name of the Tivoli Access Manager authorization server. Use the format *auth_server* **:** *port* **:** *priority*. The authorization server communication port is set at the time of Tivoli Access Manager configuration. The default port is 7136. More than one authorization server can be specified by separating the entries with commas. Having more than one authorization server configured is useful for failover and performance. The priority value is the order of authorization server use. For example: auth_server**1:**7136**:**1,auth_server**2:**7137**:**2. A priority of 1 is still required when configuring against a single authorization server.
WebSphere Application Server administrator's distinguished name	Enter the full distinguished name of the WebSphere Application Server security administrator ID, as created in the ″Creating the security administrative user″ topic in the *Securing applications and their environment* PDF. For example: cn=wasadmin,o=*organization*,c=*country*
Tivoli Access Manager user registry distinguished name suffix	For example: o=*organization*,c=*country*

Option	Description
Tivoli Access Manager administrator's user name	Enter the Tivoli Access Manager administration user ID, as created at the time of Tivoli Access Manager configuration. This ID is usually, sec_master.
Tivoli Access Manager administrator's user password	Enter the password for the Tivoli Access Manager administrator.
Tivoli Access Manager security domain	Enter the name of the Tivoli Access Manager security domain that is used to store users and groups. If a security domain is not already established at the time of Tivoli Access Manager configuration, click **Return** to accept the default.
Embedded Tivoli Access Manager listening port set	WebSphere Application Server needs to listen on a TCP/IP port for authorization database updates from the policy server. More than one process can run on a particular node and machine so a list of ports is required for the processes. Enter the ports that are used as listening ports by Tivoli Access Manager clients, separated by a comma. If you specify a range of ports, separate the lower and higher values by a colon. For example, 7999, 9990:9999.
Defer	Set to *yes*, this option defers the configuration of the management server until the next restart. Set to *no*, configuration of the management server occurs immediately. Managed servers are configured on their next restart.

3. When all information is entered, select **F** to save the configuration properties or **C** to cancel from the configuration process and discard entered information.

What to do next

Now enable the JACC provider for Tivoli Access Manager - see Enabling the JACC provider for Tivoli Access Manager topic in the *Securing applications and their environment* PDF.

Securing communications using the wsadmin tool

The application server provides several methods to secure communication between a server and a client. Use this topic to configure Secure Sockets Layer (SSL), keystores, certificate authorities, key sets and groups, and certificates.

- Configure secure communications using SSL.

  Use the SSLConfigCommands, SSLConfigGroupCommands, DynamicSSLConfigSelections and SSLTransport command groups for the AdminTask object, and complete the following tasks to create and administer SSL configurations:

  – Create an SSL configuration at the node scope using scripting.

  – Automate SSL configurations using scripting.

- Create a keystore configuration.

  Use the KeyStoreCommands command group for the AdminTask object, and complete the following tasks to create and administer keystore configurations.

  – Update default key store passwords using scripting

- Create a certificate authority (CA) client configuration.

  A CA client object contains all of the configuration information necessary to connect to a third-party CA server. Use the CAClientCommands command group for the AdminTask object, and complete the following tasks to create and administer CA client objects in your configuration:

  - Configure CA clients using scripting
  - Administer CA clients using scripting
- Administer certificate configurations.

  Use the CertificateRequestCommands, PersonalCertificateCommands, and SignerCertificateCommands command groups for the AdminTask object, and complete the following tasks to administer personal certificates, CA certificates, and self-signed certificates:
  - Create self-signed certificates using scripting
  - Configure CA certificates using scripting
- Create key sets and key groups.

  Use the KeySetCommands, KeySetGroupCommands, and KeyReferenceCommands command groups for the AdminTask object to create and administer key set and group configurations.

Creating an SSL configuration at the node scope using scripting

An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

To use the information in this task effectively, familiarize yourself with the instructions in the Creating a Secure Sockets Layer configuration topic in the *Securing applications and their environment* PDF. Perform the following task to create an Secure Socket Layer (SSL) configuration at the node scope:

1. List the existing configuration objects. Perform any of the following:
   - List some of the configuration objects that you may need when you create a new SSL configuration.

     For example, you want to see which management scopes have already been defined. If the one you need does not exist you will need to create it.
     - Using Jacl:

       ```
       $AdminTask listManagementScopes {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02}
       ```
     - Using Jython:

       ```
       AdminTask.listManagementScopes ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02]')
       ```

     This shows an existing cell scope and existing node scope that you can use. If you want to create a different scope, use the **createManagementScope** command of the AdminTask object to define a different one. The valid scope parameters are cell, nodegroup, node, server, cluster, and endpoint. See the Central management of Secure Sockets Layer configurations topic in the *Securing applications and their environment* PDF for more information on scope definitions.
   - List the key stores that exist in the configuration including key stores and trust stores.
     - Using Jacl:

       ```
       $AdminTask listKeyStores -all true
       ```
     - Using Jython:

       ```
       AdminTask.listKeyStores('-all true')
       ```

     Example output:

     ```
     CellDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1)
     CellDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_2)
     CellLTPAKeys(cells/BIRKT40Cell02|security.xml#KeyStore_3)
     ```

The previous example only lists the key stores for the default management scope which is also known as the cell scope. To obtain key stores for other scopes, specify the scopeName parameter, for example:

– Using Jacl:

```
$AdminTask listKeyStores {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 }
```

– Using Jython:

```
$AdminTask listKeyStores ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02]')
```

Example output:

```
CellDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1)
CellDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_2)
CellLTPAKeys(cells/BIRKT40Cell02|security.xml#KeyStore_3)
NodeDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1134610924357)
NodeDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_1134610924377)
```

- List specific trust or key managers. Be sure to display the object name for the trust managers. You will need the object name for the SSL configuration because you can specify multiple trust manager instances.

  – Using Jacl:

  ```
  $AdminTask listTrustManagers {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -displayObjectName true }
  ```

  – Using Jython:

  ```
  AdminTask.listTrustManagers ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -displayObjectName true]')
  ```

  Example output:

  ```
  IbmX509(cells/BIRKT40Cell02|security.xml#TrustManager_1)
  IbmPKIX(cells/BIRKT40Cell02|security.xml#TrustManager_2)
  IbmX509(cells/BIRKT40Cell02|security.xml#TrustManager_1134610924357)
  IbmPKIX(cells/BIRKT40Cell02|security.xml#TrustManager_1134610924377)
  ```

2. Create the node-scoped SSL configuration in interactive mode. Now that we have the information we need to choose from, we need to decide if these objects are sufficient or if we need to create new ones. For now, we will reuse what we've already got in the configuration and save creating new instances to task documents specific to those objects.

   - Using Jacl:

     ```
     $AdminTask createSSLConfig -interactive
     ```

   - Using Jython:

     ```
     AdminTask.createSSLConfig ('[-interactive]')
     ```

   Example output:

   ```
   Create a SSL Configuration.

   *SSL Configuration Alias (alias): BIRKT40Node02SSLConfig
   Management Scope Name (scopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
   Client Key Alias (clientKeyAlias): default
   Server Key Alias (serverKeyAlias): default
   SSL Type (type): [JSSE]
   Client Authentication (clientAuthentication): [false]
   Security Level of the SSL Configuration (securityLevel): [HIGH]
   Enabled Ciphers SSL Configuration (enabledCiphers):
   JSSE Provider (jsseProvider): [IBMJSSE2]
   Client Authentication Support (clientAuthenticationSupported): [false]
   SSL Protocol (sslProtocol): [SSL_TLS]
   Trust Manager Object Names (trustManagerObjectNames): (cells/BIRKT40Cell02|security.xml#TrustManager_1)
   *Trust Store Name (trustStoreName): NodeDefaultTrustStore
   Trust Store Scope (trustStoreScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
   *Key Store Name (keyStoreName): NodeDefaultKeyStore
   Key Store Scope Name (keyStoreScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
   Key Manager Name (keyManagerName): IbmX509
   Key Manager Scope Name (keyManagerScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02

   Create SSL Configuration
   ```

```
F (Finish)
C (Cancel)

Select [F, C]: [F] F
WASX7278I: Generated command line: $AdminTask createSSLConfig {-alias BIRKT40Node02SSLConfig -scopeName
(cell):BIRKT40Cell02:(node):BIRKT40Node02 -clientKeyAlias default -serverKeyAlias default
-trustManagerObjectNames (cells/BIRKT40Cell02|security.xml#TrustManager_1) -trustStoreName
NodeDefaultTrustStore -trustStoreScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -keyStoreName
NodeDefaultKeyStore -keyStoreScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -keyManagerName
IbmX509 -keyManagerScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 }
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Results

The name of the SSL configuration object that you created, for example, (`cells/BIRKT40Cell02|security.xml#SSLConfig_1136652770753`), appears in the `security.xml` file.

Example `security.xml` file output:

```
<repertoire xmi:id="SSLConfig_1136652770753" alias="BIRKT40Node02SSLConfig" type="JSSE"
managementScope="ManagementScope_1134610924357">
<setting xmi:id="SecureSocketLayer_1136652770924" clientKeyAlias="default" serverKeyAlias="default"
clientAuthentication="false" securityLevel="HIGH"  jsseProvider="IBMJSSE2" sslProtocol="SSL_TLS"
keyStore="KeyStore_1134610924357" trustStore="KeyStore_1134610924377" trustManager="TrustManager_1"
keyManager="KeyManager_1134610924357"/>
</repertoire>
```

What to do next

Once you create the SSL configuration object, the next step is to use it. There are several different ways that you can associate SSL configurations with protocols, for example:

- Set the SSL configuration on the thread programmatically.
- Associate the SSL configuration with an outbound protocol or a target host and port.
- Directly associating the SSL configuration using the alias.
- Centrally managing the SSL configurations by associating them with SSL configuration groups or zones so that they are used based upon the group from where the end point exists.

Automating SSL configurations using scripting

SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

AdminTask can be used in a interactive mode and batch mode. For automation the batch mode options should be used. AdminTask batch mode can be called in a JACL or Python script. Interactive mode will step through all the parameter the task needs, requires ones are marked with a ’*’. Before the interactive task executes the task it echoes the batch mode syntax of the task to the screen. This can be helpful when writing batch mode scripts.

There attributes needed to create an ssl configurations:

- A key store
- Default client certificate alias
- Default server certificate alias
- Trust store
- The handshake protocol
- The ciphers needed during handshake
- Supporting client authentication or not

If automating the creation of a SSL Configuration it may be needed to create some of the attribute values needed like the key store, trust store, key manager, and trust managers.

- To create a SSL configuration the createSSLConfig AdminTask can be used. To make changes to the SSL configurations use the modifySSLConfig AdminTask.
  - Interactive mode:

    Interactive mode steps you through all attributes and tell you the default value of the attribute if there is one. The default value is in '[]' on the prompt line. The actual flag used in batch mode is in '()' on each prompt line. If you are using the default value then the flag will not show up on the batch command line.

    Using Jacl:

    ```
    $AdminTask createSSLConfig -interactive
    ```

  - Using Jython:

    ```
    AdminTask.createSSLConfig ('[interactive]')
    ```

  Example output:

  ```
  *SSL Configuration Alias (alias): testSSLConfig
  Management Scope Name (scopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
  Client Key Alias (clientKeyAlias): clientCert
  Server Key Alias (serverKeyAlias): serverCert
  SSL Type (type): [JSSE]
  Client Authentication (clientAuthentication): [false]
  Security Level of the SSL Configuration (securityLevel): [HIGH] HIGH
  Enabled Ciphers SSL Configuration (enabledCiphers):
  JSSE Provider (jsseProvider): [IBMJSSE2]
  Client Authentication Support (clientAuthenticationSupported): [false]
  SSL Protocol (sslProtocol): [SSL_TLS] SSL_TLS
  Trust Manager Object Names (trustManagerObjectNames):
  *Trust Store Name (trustStoreName): testTrustStore
  Trust Store Scope (trustStoreScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
  *Key Store Name (keyStoreName): testKeyStore
  Key Store Scope Name (keyStoreScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
  Key Manager Name (keyManagerName): IbmX509
  Key Manager Scope Name (keyManagerScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01

  Create SSL Configuration

  F (Finish)
  C (Cancel)

  Select [F, C]: [F]
  WASX7278I: Generated command line: $AdminTask createSSLConfig {-alias testSSLConfig
  -scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
  -serverKeyAlias serverCert -trustStoreName testTrustStore
  -trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
  -keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
  -keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01 }
  (cells/HOSTNode01Cell|security.xml#SSLConfig_1137687301834)
  ```

  At the end of the output, the batch mode parameters are provided.

  - Batch mode:

Using Jacl:

```
$AdminTask createSSLConfig {-alias testSSLConfig
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
-serverKeyAlias serverCert -trustStoreName testTrustStore
-trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01}
```

– Using Jython:

```
AdminTask.createSSLConfig ('[-alias testSSLConfig
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
-serverKeyAlias serverCert -trustStoreName testTrustStore
-trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01]')
```

Example output:

```
(cells/HOSTNode01Cell|security.xml#SSLConfig_1137687301834)
```

- Key Stores and Trust Stores The key store and trust store may already exist or a new one may need to be created. To create a new key store or trust store use the createKeyStore AdminTask. It will create a key store file and store the configuration object in the system configuration. A trust store is just a key store that usually only has signer certificates in it. To create a key store enter:

  – Using Jacl:

  ```
  $AdminTask createKeyStore {-keyStoreName testKeyStore  -keyStoreType PKCS12
  -keyStoreLocation $(USER_INSTALL_ROOT)\testKeyStore.p12 -keyStorePassword abcd
  -keyStorePasswordVerify abcd  -keyStoreIsFileBased true -keyStoreReadOnly false}
  ```

  – Using Jython:

  ```
  AdminTask.createKeyStore ('[-keyStoreName testKeyStore  -keyStoreType PKCS12
  -keyStoreLocation $(USER_INSTALL_ROOT)\testKeyStore.p12 -keyStorePassword abcd
  -keyStorePasswordVerify abcd  -keyStoreIsFileBased true -keyStoreReadOnly false]')
  ```

  To populate the key store with certificates see “Managing Certificates using AdminConsole and Admin Task” The key store and trust store are required to create a SSL configuration. Use the ’-keyStoreName’ and ’-trustStoreName’ flags on the createSSLConfig. There scopes can be added with the ’-keyStoreScope’ flag and ’-trustStoreScope’ flags.

- Key Manager Key manager are used to determine how a certificate is selected. The IbmX509 key manager is in the security configuration by default. If a different key manager is needed then use createKeyManager AdminTask to create it. To create a key manager enter:

  – Using Jacl:

  ```
  $AdminTask createKeyManager {-name testKeyManager
  -scopeName  (cell):HOSTNode01Cell:(node):HOSTNode01
  -provider IBMJSSE2 -algorithm specialAlgorithm }
  ```

  – Using Jython:

  ```
  AdminTask.createKeyManager ('[-name testKeyManager
  -scopeName  (cell):HOSTNode01Cell:(node):HOSTNode01
  -provider IBMJSSE2 -algorithm specialAlgorithm]')
  ```

  To supply a key manager on the createSSLConfig AdminTask use the ’-keyManagerName’ along with the ’-keyManagerScope’ flag.

- Trust Manager Trust managers are use to determine how trust is established during ssl communication. The IbmX509 and IbmPKIX are trust managers are in the security configuration by default. If a different or additional trust manager is needed then use the createTrustManger AdminTask to create it. To create a trust manager enter:

  – Using Jacl:

  ```
  $AdminTask createTrustManager {-name testTrustManager
  -scopeName (cell):HOSTNode01Cell:(node):HOSTNode01
  -provider IBMJSSE2 -algorithm specialAlgorithm }
  ```

  – Using Jython:

```
AdminTask.createTrustManager ('[-name testTrustManager
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-provider IBMJSSE2 -algorithm specialAlgorithm]')
```

The SSL Configuration can have multiple trust managers. To supply multiple trust managers give a comma separated list of the trust managers configuration IDs with the `-trustManagerObjectNames` flag. When you create a trust manager the configuration object ID is returned. To get a list of trust managers object IDs use the **listTrustManagers** command of the AdminTask object with the `-displayObjectName true` flag. For example:

```
wsadmin>$AdminTask listTrustManagers -interactive
List Trust Managers

List trust managers.

Management Scope Name (scopeName):
Display list in ObjectName Format (displayObjectName): [false] true

List Trust Managers

F (Finish)
C (Cancel)

Select [F, C]: [F]
Inside generate script command
WASX7278I: Generated command line: $AdminTask listTrustManagers {-displayObjectName true }
IbmX509(cells/IBM-0AF8DABCF16Node01Cell|security.xml#TrustManager_IBM-0AF8DABCF16Node01_1)
IbmPKIX(cells/IBM-0AF8DABCF16Node01Cell|security.xml#TrustManager_IBM-0AF8DABCF16Node01_2)
```

Updating default key store passwords using scripting

Use the Jython or Jacl scripting language to change the default key store passwords. A key store file is created with a default password when you install the application server. Change this password to protect your security configuration.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

When you install the application server, each server creates a key store and trust store for the default SSL configuration with the default password `WebAS`. To protect the security of the key store files and the SSL configuration, you must change the password. The following examples update the default password:

- Change multiple key stores passwords. The **changeMultipleKeyStorePasswords** command updates all of the key stores that have the same password. For example:
  - Using Jacl:

    ```
    $AdminTask changeMultipleKeyStorePasswords  {-keyStorePassword WebAS
    -newKeyStorePassword  secretPwd   -newKeyStorePasswordVerify secretPwd}
    ```

  - Using Jython:

    ```
    AdminTask.changeMultipleKeyStorePasswords  ['(-keyStorePassword WebAS
    -newKeyStorePassword secretPwd -newKeyStorePasswordVerify secretPwd]')
    ```

- Change the password of a single key store. The **changeKeyStorePassword** command updates the password of an individual key store. For example:
  - Using Jacl:

    ```
    $AdminTask changeKeyStorePassword {-keyStoreName testKS
    -keyStoreScope (cell):localhost:(server):server1
    -keyStorePassword WebAS -newKeyStorePassword secretPwd
    -newKeyStorePasswordVerify secretPwd}
    ```

– Using Jython:

```
AdminTask.changeKeyStorePassword ('[-keyStoreName testKS
-keyStoreScope (cell):localhost:(server):server1
-keyStorePassword WebAS -newKeyStorePassword secretPwd
-newKeyStorePasswordVerify secretPwd]')
```

Configuring certificate authority client objects using the wsadmin tool

Use this topic to create a certificate authority (CA) client object. The client object contains all of the configuration information necessary to connect to your third-party CA server. A CA client must exist in your configuration before you can issue a request to the CA to create personal certificates with the requestCACertificate command.

Before you begin

A CA client object contains information that the system uses to connect to a certificate authority. Implement the com.ibm.ws.WSPKIClient interface to connect to the certificate authority and provide the com.ibm.ws.WSPKIClient class when creating the CA client object.

About this task

If a CA client does not exist in your configuration, use the steps in this topic to create a new CA client.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine if a CA client exists in your configuration.

   Use the following listCAClients command to list all certificate authority clients in your configuration:

```
print AdminTask.listCAClients()
```

3. If no CA clients exist, then create a new CA client.

   Use the createCAClient command to create a new CA client object. The application server connects to a CA server through the WSPKIClient() implementation, which handles all connections and communications with the CA server. You must specify the following configuration information for a new CA client object:

Parameter	Description	Data Type
-CAClientName	Specify a name to uniquely identify the CA client object.	String

You can specify additional configuration information using the following parameters:

Parameter	Description	Data Type
-scopeName	Specify the management scope of the CA client. For a deployment manager profile, the system uses the cell scope as the default value. For an application server profile, the system uses the node scope as the default value.	String
-pkiClientImplClass	Specify the class path that implements the WSPKIClient interface. The system uses this path to connect to the CA and to issue requests to the CA. The default value is `com.ibm.wsspi.ssl.WSPKIClient`.	String
-host	Specify the host name in your system where the CA resides.	String
-port	Specify the port on the server where the CA listens.	String
-userName	Specify the user name to use to authenticate to the CA.	String
-password	Specify the password for the user name that authenticates to the CA.	String
-frequencyCheck	Specify how often, in minutes, the system checks with the CA to determine if a certificate has been created.	String

Parameter	Description	Data Type
-retryCheck	Specify the number of times to check with the CA to determine if a certificate has been created.	String
-customProperties	Specifies a comma separated list of attribute and value (attribute=value) custom property pairs to add to the CA client object.	String

Use the following example command to create a new CA client object:

```
AdminTask.createCAClient('[-caClientName clientObj01 -pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient
-host machine011 -port 9022 -userName admin -password pw4admin]')
```

The command returns the object name of the CA client that has been created.

4. Save your configuration changes.

What to do next

If the CA client object was successfully created, then you can configure the application server to use a personal certificate created by an external CA.

Administering certificate authority clients using the wsadmin tool

Use this topic to modify certificate authority (CA) client objects. The client object contains all of the configuration information necessary to connect to your third-party CA server.

Before you begin

You must configure a CA client object in your environment.

About this task

For existing CA client objects, use the steps in this topic to view, modify, or delete existing CA client object configurations.

- View existing CA client objects and configuration data. Use the listCAClients and getCAClient commands to query your environment for your existing CA clients.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List all CA client objects in your configuration.

     Use the listCAClients command to list all certificate authority clients in your configuration. If you do not provide a value for the -scopeName parameter, then the command queries the cell if you use a deployment manager profile or queries the node if you use an application server profile. Use the -all parameter to query your environment without using a specific scope, as the following example demonstrates:

```
print AdminTask.listCAClients('-all true')
```

     The command returns an array of attribute lists, displaying one attribute list for each CA client, as the following example output displays:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name jenCAClient] [baseDn ] [_Websphere_Config_Da
ta_Id cells/myCell01|security.xml#CAClient_1181834566881] [port 2950] [CACertifi
cate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
e_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pa
ssword ] [host ] ]'
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphere_Config_Dat
a_Id cells/myCell01|security.xml#CAClient_1181834566882] [port 2951] [CACertific
ate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pas
sword ] [host ] ]'
```

  3. List the configuration attributes for a specific CA client.

     Use the getCAClient command to view the list of attributes for a specific CA client, as the following example demonstrates:

```
print AdminTask.getCAClient('-caClientName myCAClient')
```

The command returns an attribute list that contains the attribute and value pairs for the specific CA client, as the following example demonstrates:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementSc
ope_1)] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphe
re_Config_Data_Id cells/myCell01|security.xml#CAClient_1181834566882] [por
t 2951] [CACertificate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [u
serId ] [_Websphere_Config_Data_Type CAClient] [retryCheck 0] [properties ] [fre
quencyCheck 0] [password ] [host ] ]'
```

- Modify your existing CA client object configuration data. Use the modifyCAClient command to change one or more configuration attributes for a specific CA client.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine which configuration attributes to edit.

     The modifyCAClient modifies all attributes that you specify with the command parameters. If you do not specify a parameter, then its corresponding attribute does not change. You can edit the following configuration data with the modifyCAClient command:

Parameter	Description	Data Type
-scopeName	Specify the management scope of the CA client. For a deployment manager profile, the system uses the cell scope as the default. For an application server profile, the system uses the node scope as the default.	String
-pkiClientImplClass	Specify the class path that implements the WSPKIClient interface. The system uses this path to connect to the CA and to issue requests to the CA.	String
-host	Specify the host name in your system where the CA resides.	String
-port	Specify the port on the server where the CA listens.	String
-userName	Specify the user name to use to authenticate to the CA.	String
-password	Specify the password for the user name that authenticates to the CA.	String
-frequencyCheck	Specify how often, in minutes, the system should check with the CA to determine if a certificate has been created.	String
-retryCheck	Specify the number of times to check with the CA to determine if a certificate has been created.	String
-customProperties	Specifies a comma separated list of attribute and value (attribute=value) custom property pairs to modify on the CA Client object. You can create, modify, or remove properties. To remove a property specify `attribute=` attribute as equal to no value.	String

  3. Modify specific configuration attributes for a CA client object.

     Use the following example command to modify the port number of the CA, the user name, and password attributes for the `myCAClient` CA client object:

```
AdminTask.modifyCAClient('[-caClientName myCAClient -port 4060 -userName admin -password password4admin -pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient]')
```

  4. Save your configuration changes.
- Remove a CA client object from your configuration. Use the deleteCAClient command to delete a CA client object from your configuration. The command does not delete the CA client object if the CA client to delete is referenced by a certificate object.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the CA client object to delete.

     Use the listCAClients command to list all certificate authority clients in your configuration. If you do not provide a value for the -scopeName parameter, then the command queries the cell if you use a

deployment manager profile or queries the node if you use an application server profile. Use the -all parameter to query your environment without using a specific scope, as the following example demonstrates:

```
print AdminTask.listCAClients('-all true')
```

3. Delete the CA client object of interest.

   Use the deleteCAClient command to delete the CA client object from your configuration. Use the -caClientName parameter to specify the CA client to delete. You can optionally specify the management scope of the CA client object with the scopeName parameter. The following example command removes the `myCAClient` CA client object:

```
AdminTask.deleteCAClient('[-caClientName myCAClient]')
```

   If you receive an error message, then verify that the CA client object of interest exists in your configuration and that it is not referenced by a certificate object in your security configuration.

4. Save your configuration changes.

Setting a certificate authority certificate as the default certificate using the wsadmin tool

Use this topic to make a request to an external certificate authority (CA) to create a personal certificate. After the CA returns the certificate and the certificate is saved in the keystore, then you can use it as the server default personal certificate.

Before you begin

You must configure a CA client object in your environment. The client object contains all of the configuration information necessary to connect to your third-party CA server.

About this task

After profile creation, the system is assigned a default chained personal certificate. Use the following steps to modify the application server to use a default personal certificate created by an external CA.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Verify that a certificate authority client exists in your configuration. Use the listCAClients command to query your environment for all existing certificate authority clients and configuration attributes, or the getCAClient command to return the configuration attributes for a specific certificate authority client. If the listCAClients or getCAClient commands do not return any attributes, then you must create a certificate authority client object before you can complete the remaining steps.
   - List all certificate authority client objects in your configuration.

     Use the listCAClients command to list all certificate authority clients in your configuration. If you do not provide a value for the -scopeName parameter, then the command queries the cell if you use a deployment manager profile or queries the node if you use an application server profile. Use the -all parameter to query your environment without using a specific scope, as the following example demonstrates:

```
print AdminTask.listCAClients('-all true')
```

     The command returns an array of attribute lists, displaying one attribute list for each CA client, as the following example output displays:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name jenCAClient] [baseDn ] [_Websphere_Config_Da
ta_Id cells/myCell01|security.xml#CAClient_1181834566881] [port 2950] [CACertifi
cate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
e_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pa
ssword ] [host ] ]'
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphere_Config_Dat
a_Id cells/myCell01|security.xml#CAClient_1181834566882] [port 2951] [CACertific
ate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pas
sword ] [host ] ]'
```

- List the configuration attributes for a specific certificate authority client.

  Use the getCAClient command to view the list of attributes for a specific certificate authority client, as the following example demonstrates:

```
print AdminTask.getCAClient('-caClientName myCAClient')
```

  The command returns an attribute list that contains the attribute and value pairs for the specific certificate authority client, as the following example demonstrates:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementSc
ope_1)] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphe
re_Config_Data_Id cells/myCell01|security.xml#CAClient_1181834566882] [por
t 2951] [CACertificate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [u
serId ] [_Websphere_Config_Data_Type CAClient] [retryCheck 0] [properties ] [fre
quencyCheck 0] [password ] [host ] ]'
```

3. Optional: If a certificate authority client does not exist in your environment, then configure a CA client object.
4. Optional: View the current default personal certificate.

   Use the following listPersonalCertificates command to display the current default personal certificate to replace:

```
AdminTask.listPersonalCertificates('[-keyStoreName CellDefaultKeyStore -keyStoreScope (cell):myCell01]')
```

5. Request a certificate from a certificate authority.

   Before the current default personal certificate can be replaced, you must request a certificate from a certificate authority. You can create a new certificate request or use the createCertificateRequest command to use a predefined certificate request. The system uses the certificate request and the certificate authority configuration information from the CA client object to request the certificate from the certificate authority. If the certificate authority returns a certificate, then the requestCAcertificate command stores the certificate in the specified key store and returns a message of `COMPLETE`. Use the requestCACertificate command and the following required parameters to request a certificate from a certificate authority:

Parameter	Description	Data Type
-certificateAlias	Specifies the alias of the certificate. You can specify a predefined certificate request.	String
-keyStoreName	Specifies the name of the keystore object that stores the CA certificate. Use the listKeyStores command to display a list of available keystores.	String
-caClientName	Specifies the name of the CA client that was used to create the CA certificate.	String
-revocationPassword	Specifies the password to use to revoke the certificate at a later date.	String

   You can also use the following parameters to specify additional certificate request options. If you do not specify an optional parameter, then the command uses the default value.

Parameter	Description	Data Type
-keyStoreScope	Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope.	String
-caClientScope	Specifies the management scope of the CA client. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope.	String
-certificateCommonName	Specifies the common name (CN) part of the full distinguished name (DN) of the certificate. This common name can represent a person, company, or machine. For Web sites, the common name is frequently the DNS host name where the server resides.	String
-certificateSize	Specifies the size of the certificate key. The valid values are `512`, `1024`, and `2048`. The default value is `1024`.	String

Parameter	Description	Data Type
-certificateOrganization	Specifies the organization portion of the distinguished name.	String
-certificateOrganizationalUnit	Specifies the organizational unit portion of the distinguished name.	String
-certificateLocality	Specifies the locality portion of the distinguished name.	String
-certificateState	Specifies the state portion of the distinguished name.	String
-certificateZip	Specifies the zip code portion of the distinguished name.	String
-certificateCountry	Specifies the country portion of the distinguished name.	String

Use the following example command syntax to request a certificate from a certificate authority:

```
AdminTask.requestCACertificate('-certificateAlias newCertificate -keyStoreName
CellDefaultKeyStore -caClientName myCAClient -revocationPassword revokeCApw
-pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient')
```

The command returns one of two values: `Certificate COMPLETE` or `certificate PENDING`. If the command returns the `Certificate COMPLETE` message, the certificate authority returned the requested certificate and the default personal certificate is replaced. If the command returns the `certificate PENDING` message, the certificate authority did not yet return a certificate. Use the queryCACertificate command to view the current status of the certificate request, as the following example demonstrates:

```
AdminTask.queryCACertificate('-certificateAlias newCertificate -keyStoreName
CellDefaultKeyStore -pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient')
```

6. Replace the server default personal certificate.

   Use the following replaceCertificate command example to replace the existing default personal certificate with the newly created CA personal certificate:

```
AdminTask.replaceCertificate('-keyStoreName CellDefaultKeyStore -certificateAlias
defaultPersonalCertificate -replacementCertificateAlias newCertificate')
```

7. Save your configuration changes.

Results

The default personal certificate for the server is a certificate that is created by an external CA.

What to do next

If the CA client object was successfully created, then you can configure the application server to use a personal certificate created by an external CA.

Creating certificate authority (CA) personal certificates using the wsadmin tool

Use this topic to create CA certificates from a certificate authority (CA).

Before you begin

You must configure a CA client object in your environment. The client object contains all of the configuration information necessary to connect to your third-party CA server.

About this task

Use the following information to create a CA personal certificate using a CA client.

1. Optional: Query your configuration for keystores to determine where system stores the new CA certificate.

Use the listKeyStores command to list all keystores for a specific management scope. Specify the -scopeName parameter to display keystores within a specific management scope, or set the -all parameter to `true` to display all keystores regardless of scope. The following example lists all keystores in your configuration:

```
AdminTask.listKeystores('-all true')
```

The command returns the following sample output:

```
CellDefaultKeyStore(cells/myCell|security.xml#KeyStore_1)
CellDefaultTrustStore(cells/myCell|security.xml#KeyStore_2)
CellLTPAKeys(cells/myCell|security.xml#KeyStore_3)
NodeDefaultKeyStore(cells/myCell|security.xml#KeyStore_1598745926544)
NodeDefaultTrustStore(cells/myCell|security.xml#KeyStore_1476529854789)
```

Use the getKeyStoreInfo command and specify the -keyStoreName parameter to return additional information about the keystore of interest, as the following example displays:

```
AdminTask.getKeyStoreInfo('[-keyStoreName CellDefaultKeyStore]')
```

The command returns the following configuration information for the keystore of interest:

```
[ [location ${CONFIG_ROOT}/cells/myCell/key.p12] [password *****] [_Websphere
e_Config_Data_Id cells/myCell|security.xml#KeyStore_1] [_Websphere_Config_Da
ta_Version ] [useForAcceleration false] [slot 0] [type PKCS12] [additionalKeySto
reAttrs ] [fileBased true] [_Websphere_Config_Data_Type KeyStore] [customProvide
rClass ] [hostList ] [createStashFileForCMS false] [description [Default key sto
re for JenbCell01]] [readOnly false] [initializeAtStartup false] [managementScop
e (cells/JenbCell01|security.xml#ManagementScope_1)] [usage SSLKeys] [provider I
BMJCE] [name CellDefaultKeyStore] ]
```

2. Optional: Determine which CA client to use.

   Use the listCAClients command to list the CA clients that exist in your configuration. Specify the -scopeName parameter to display CA clients within a specific management scope, or set the -all parameter to `true` to display all CA clients regardless of scope. The following example lists all CA clients in your configuration:

```
AdminTask.listCAClients('-all true')
```

3. Create a CA personal certificate.

   Use the requestCACertificate command to create a new CA personal certificate in your environment. The system uses the certificate request and the certificate authority configuration information from the CA client object to request the certificate from the certificate authority. If the certificate authority returns a certificate, the requestCAcertificate command stores the certificate in the specified key store and returns a message of `COMPLETE`. Use the requestCACertificate command and the following required parameters to request a certificate from a certificate authority:

Parameter	Description	Data type
-certificateAlias	Specifies the alias of the certificate. You can specify a predefined certificate request.	String
-keyStoreName	Specifies the name of the keystore object that stores the CA certificate. Use the listKeyStores command to display a list of available keystores.	String
-caClientName	Specifies the name of the CA client that was used to create the CA certificate.	String
-revocationPassword	Specifies the password to use to revoke the certificate at a later date.	String

You can also use the following parameters to specify additional certificate request options. If you do not specify an optional parameter, the command uses the default value.

Parameter	Description	Data type
-keyStoreScope	Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope.	String

Parameter	Description	Data type
-caClientScope	Specifies the management scope of the CA client. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope.	String
-certificateCommonName	Specifies the common name (CN) part of the full distinguished name (DN) of the certificate. This common name can represent a person, company, or machine. For Web sites, the common name is frequently the DNS host name where the server resides.	String
-certificateSize	Specifies the size of the certificate key. The valid values are 512, 1024, and 2048. The default value is 1024.	String
-certificateOrganization	Specifies the organization portion of the distinguished name.	String
-certificateOrganizationalUnit	Specifies the organizational unit portion of the distinguished name.	String
-certificateLocality	Specifies the locality portion of the distinguished name.	String
-certificateState	Specifies the state portion of the distinguished name.	String
-certificateZip	Specifies the zip code portion of the distinguished name.	String
-certificateCountry	Specifies the country portion of the distinguished name.	String

Use the following example command syntax to request a certificate from a certificate authority:

```
AdminTask.requestCACertificate('-certificateAlias newCertificate -keyStoreName CellDefaultKeyStore -CAClientName myCAClient -revocationPassword revokeCApw')
```

The command returns one of two values: `Certificate COMPLETE` or `certificate PENDING`. If the command returns the `Certificate COMPLETE` message, the certificate authority returned the requested certificate and the default personal certificate is replaced. If the command returns the `certificate PENDING` message, the certificate authority did not yet return a certificate. Use the queryCACertificate command to view the current status of the certificate request, as the following example displays:

```
AdminTask.queryCACertificate('-certificateAlias newCertificate -keyStoreName CellDefaultKeyStore')
```

4. Save your configuration changes.

Results

The default personal certificate for the server is a certificate that is created by an external CA.

Revoking certificate authority personal certificates using the wsadmin tool

You can revoke CA certificates from a certificate authority (CA). Revoke personal certificates that are no longer being used in your configuration.

Before you begin

Use the requestCACertificate command to create a personal certificate with the requestCACertificate task before you can request that the certificate authority revoke the certificate. Certificates created with the requestCACertificate command have an associated reference object in the configuration that you can use to submit the certificate revocation request to the certificate authority.

About this task

This topic uses the revokeCACertificate command to submit a request to revoke a certificate on the certificate authority. You can only revoke a certificate that was created with the requestCACertificate command. You must specify the revocation password that was provided when the certificate was created.

Use the same password to revoke the certificate on the certificate authority.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the CA personal certificate to revoke.

   Use the listPersonalCertificates command to view a list of all personal certificates and associated attributes for a specific keystore, as the following example demonstrates:

```
AdminTask.listPersonalCertificates('-keyStoreName CellDefaultKeyStore')
```

   The command returns an attribute list for each personal certificate, including CA personal certificates. CA personal certificates only return the status attribute. You can revoke each CA personal certificates that returns a `COMPLETE` status. Determine which CA personal certificate to revoke.
3. Revoke a CA personal certificate.

   Use the revokeCACertificate command to revoke the CA personal certificate of interest. You must specify the name of the keystore, certificate alias, and revocation password using the following parameters:

Parameter	Description	Data Type
-keyStoreName	Specifies the name of the keystore where the CA personal certificate is stored.	String
-certificateAlias	Specifies the unique name that identifies the CA personal certificate object and the alias name of the certificate in the keystore.	String
-revocationPassword	Specifies the password needed to revoke the certificate. This is the same password that was provided when the certificate was created.	String

   You can specify additional information with the following optional parameters:

Parameter	Description	Data Type
-keyStoreScope	Specifies the management scope of the keystore. For a deployment manager profile, the system uses the cell scope as the default value. For an application server profile, the system uses the node scope as the default value.	String
-revocationReason	Specifies the reason for revoking the certificate of interest. The default value for this parameter is `unspecified`.	String

   The following example revokes a CA personal certificate:

```
AdminTask.revokeCACertificate('[-keyStoreName CellDefaultKeyStore -certificateAlias myCertificate -revocationPassword pw4revoke]')
```

4. Save your configuration changes.

CAClientCommands command group for the AdminTask object

You can use the Jython scripting language to manage your certificate authority (CA) client configurations with the wsadmin tool. Use the commands and parameters in the CAClientCommands group to create, modify, query, and remove connections to a third-party CA server.

Use the following commands to manage your certificate authority (CA) client configurations:

- “createCAClient” on page 646
- “modifyCAClient” on page 647
- “getCAClient” on page 648
- “deleteCAClient” on page 648
- “listCAClients” on page 649

createCAClient

The createCAClient command creates a new CA client object in your configuration. The application server connects to a CA server through the WSPKIClient() implementation, which handles all connections and communications with the CA server.

Target object

None.

Required parameters

-caClientName
Specifies a name to uniquely identify the CA client object. (String, required)

-pkiClientImplClass
Specifies the class path that implements the WSPKIClient interface. The system uses this path to connect to the CA and to issue requests to the CA. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the CA client. For a deployment manager profile, the system uses the cell scope as the default value. For an application server profile, the system uses the node scope as the default value. (String, optional)

-host
Specifies the host name in your system where the CA resides. (String, optional)

-port
Specifies the port on the server where the CA listens. (String, optional)

-userName
Specifies the user name to use to authenticate to the CA. (String, optional)

-password
Specifies the password for the user name that authenticates to the CA. (String, optional)

-frequencyCheck
Specifies how often, in minutes, the system communicates with the CA to determine if a certificate has been created. (String, optional)

-retryCheck
Specifies the number of times to communicate with the CA to determine if a certificate has been created. (String, optional)

-customProperties
Specifies a comma-separated list of attribute and value custom property pairs to add to the CA client object, using the following format: `attribute=value,attribute=value`. (String, optional)

Return value

The command returns the object name of the CA client that the system creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.createCAClient('[-caClientName clientObj01 -pkiClientImplClass
com.ibm.wsspi.ssl.WSPKIClient -host machine011 -port 9022
-userName admin -password pw4admin]')
```

- Using Jython list:

```
AdminTask.createCAClient(['-caClientName', 'clientObj01', '-pkiClientImplClass',
'com.ibm.wsspi.ssl.WSPKIClient', '-host', 'machine011', '-port', '9022',
'-userName', 'admin', '-password', 'pw4admin'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createCAClient('-interactive')
```

modifyCAClient

The modifyCAClient command modifies your existing CA client object configuration data. You can modify one or multiple configuration attributes for a specific CA client.

Target object

None.

Required parameters

-caClientName
Specifies the name of the CA client of interest. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the CA client. For a deployment manager profile, the system uses the cell scope as the default. For an application server profile, the system uses the node scope as the default. (String, optional)

-pkiClientImplClass
Specifies the class path that implements the WSPKIClient interface. The system uses this path to connect to the CA and to issue requests to the CA. (String, optional)

-host
Specifies the host name in your system where the CA resides. (String, optional)

-port
Specifies the port on the server where the CA listens. (String, optional)

-userName
Specifies the user name to use to authenticate to the CA. (String, optional)

-password
Specifies the password for the user name that authenticates to the CA. (String, optional)

-frequencyCheck
Specifies how often, in minutes, the system should check with the CA to determine if a certificate has been created. (String, optional)

-retryCheck
Specifies the number of times to check with the CA to determine if a certificate has been created. (String, optional)

-customProperties
Specifies a comma separated list of attribute and value (attribute=value) custom property pairs to modify on the CA Client object. You can create, modify, or remove properties. To remove a property specify the attribute and value as `attribute=`. (String, optional)

Return value

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyCAClient('[-caClientName myCAClient -port 4060
-userName admin -password password4admin]')
```

- Using Jython list:

```
AdminTask.modifyCAClient(['-caClientName', 'myCAClient', '-port', '4060',
'-userName', 'admin', '-password', 'password4admin'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyCAClient('-interactive')
```

getCAClient

The getCAClient command displays a list of attributes for a specific CA client.

Target object

None.

Required parameters

-caClientName
Specifies the CA client name of interest. (String, required)

Optional parameters

-scopeName
Specifies the management scope of CA client of interest. (String, optional)

Return value

The command returns an attribute list that contains the attribute and value pairs for the specific CA client, as the following example displays:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementSc
ope_1)] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphe
re_Config_Data_Id cells/myCell01|security.xml#CAClient_1181834566882] [por
t 2951] [CACertificate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [u
serId ] [_Websphere_Config_Data_Type CAClient] [retryCheck 0] [properties ] [fre
quencyCheck 0] [password ] [host ] ]'
```

Batch mode example usage

- Using Jython string:

```
print AdminTask.getCAClient('-caClientName myCAClient')
```

- Using Jython list:

```
print AdminTask.getCAClient('-caClientName', 'myCAClient')
```

Interactive mode example usage

- Using Jython string:

```
print AdminTask.getCAClient('-interactive')
```

deleteCAClient

The deleteCAClient command removes the CA client object of interest from your configuration. Use the -caClientName parameter to specify the CA client to delete. You can optionally specify the management scope of the CA client object with the scopeName parameter.

Target object

None.

Required parameters

-caClientName
Specifies the name of the CA client of interest. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the CA client of interest. (String, optional)

Return value

The command does not return output if the system successfully removes the CA client of interest. If you receive an error message, verify that the CA client object of interest exists in your configuration and that it is not referenced by a certificate object in your security configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteCAClient('[-caClientName myCAClient]')
```

- Using Jython list:

```
AdminTask.deleteCAClient(['-caClientName', 'myCAClient'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteCAClient('-interactive')
```

listCAClients

The listCAClients command lists all CA clients in your configuration or within a specific scope. If you do not provide a value for the -scopeName parameter, the command queries the cell if you use a deployment manager profile or queries the node if you use an application server profile. Use the -all parameter to query your environment without using a specific scope.

Target object

None.

Optional parameters

-scopeName
Specifies the management scope to search for CA clients. (String, optional)

-all
Specifies whether the system queries for CA clients without a specific scope. (Boolean, optional)

Return value

The command returns an array of attribute lists, displaying one attribute list for each CA client, as the following example output displays:

```
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name jenCAClient] [baseDn ] [_Websphere_Config_Da
ta_Id cells/myCell01|security.xml#CAClient_1181834566881] [port 2950] [CACertifi
cate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
e_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pa
ssword ] [host ] ]'
'[ [backupCAs ] [managementScope (cells/myCell01|security.xml#ManagementScope_1)
] [scopeName (cell):myCell01] [name myCAClient] [baseDn ] [_Websphere_Config_Dat
a_Id cells/myCell01|security.xml#CAClient_1181834566882] [port 2951] [CACertific
ate ] [pkiClientImplClass com.ibm.wsspi.ssl.WSPKIClient] [userId ] [_Websphere
_Config_Data_Type CAClient] [retryCheck 0] [properties ] [frequencyCheck 0] [pas
sword ] [host ] ]'
```

Batch mode example usage

- Using Jython string:

```
print AdminTask.listCAClients('-all true')
```

- Using Jython list:

```
print AdminTask.listCAClients('-all', 'true')
```

Interactive mode example usage

- Using Jython:

```
print AdminTask.listCAClients('-interactive')
```

Creating self-signed certificates using scripting

Use the Jython or Jacl scripting language to create self-signed certificates with the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

You can create self-signed certificates using the scripting and the AdminTask object. You can run the commands in interactive or batch mode. Interactive mode provides a way to discover the flags that you need to run the task in batch mode.

Certificates reside inside of key stores. To run the commands, you will need the name of the key store to be supplied. Use the **listKeyStore** command of the AdminTask object to get a list of key stores. If you need a new key store, use the **createKeyStore** command of the AdminTask object.

To create a personal key store, use the following examples:

- Interactive mode:
  - Using Jython:

    ```
    AdminTask.createSelfSignedCertificate ('[-interactive]')
    ```

  - Using Jacl:

    ```
    $AdminTask createSelfSignedCertificate -interactive
    ```

    Example output:

    ```
    *Key Store Name (keyStoreName): keyStore
    Key Store Scope Name (keyStoreScope):
    *Certificate Alias (certificateAlias): newCert
    "Certificate Version" (certificateVersion): 3
    *Key Size (certificateSize): [1024]
    *Common Name (certificateCommonName): localhost
    *Organization (certificateOrganization): workgroup
    Organizational Unit (certificateOrganizationalUnit): testing
    certLocality (certificateLocality): austin
    State (certificateState): Texas
    Zip (certificateZip): 78757
    Country (certificateCountry): [US]
    Validity Period (certificateValidDays): [365]
    Create Self-Signed Certificate

    F (Finish)
    C (Cancel)

    Select [F, C]: [F]

    WASX7278I: Generated command line: $AdminTask createSelfSignedCertificate
    ```

```
{-keyStoreName keyStore -certificateAlias newCert -certificateVersion 3
-certificateCommonName localhost -certificateOrganization ibm
-certificateOrganizationalUnit testing -certificateLocality austin
-certificateState Texas -certificateZip 78757 }
true
```

At the end of the output, the batch mode parameters are provided.

- Batch mode:
  - Using Jython:

```
AdminTask.createSelfSignedCertificate ('[-keyStoreName keyStore
-certificateAlias newCert -certificateVersion 3 -certificateSize 1024
-certificateCommonName localhost -certificateOrganization ibm
-certificateOrganizationalUnit testing -certificateLocality austin
-certificateState Texas -certificateZip 78757]')
```

  - Using Jacl:

```
$AdminTask createSelfSignedCertificate {-keyStoreName keyStore
-certificateAlias newCert -certificateVersion 3 -certificateSize 1024
-certificateCommonName localhost -certificateOrganization ibm
-certificateOrganizationalUnit testing -certificateLocality austin
-certificateState Texas -certificateZip 78757 }
```

keyManagerCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security. The commands and parameters in the keyManagerCommands group can be used to manage key manager settings. You can use these commands to create, modify, list, or obtain information about key managers.

The keyManagerCommands command group for the AdminTask object includes the following commands:

- "deleteKeyManager"
- "getKeyManager" on page 652
- "listKeyManagers" on page 652
- "modifyKeyManager" on page 653

deleteKeyManager

The **deleteKeyManager** command deletes the key manager settings from the configuration.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key manager. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteKeyManager {-name testKM}
```

- Using Jython string:

```
AdminTask.deleteKeyManager('[-name testKM]')
```

- Using Jython list:

```
AdminTask.deleteKeyManager(['-name', 'testKM'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteKeyManager {-interactive}
```

- Using Jython:

```
AdminTask.deleteKeyManager('-interactive')
```

getKeyManager

The **getKeyManager** command displays a properties object that contains the key manager attributes and values.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key manager. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getKeyManager {-name testKM}
```

- Using Jython string:

```
AdminTask.getKeyManager('[-name testKM]')
```

- Using Jython list:

```
AdminTask.getKeyManager(['-name', 'testKM'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getKeyManager {-interactive}
```

- Using Jython:

```
AdminTask.getKeyManager('-interactive')
```

listKeyManagers

The **listKeyManagers** command lists the key managers within a particular management scope.

Target object

None.

Required parameters

None.

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-displayObjectName
Set the value of this parameter to `true` to list the key manager objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key manager name and the management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all key managers. This parameter overrides the scopeName parameter. The default value is `false`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listKeyManagers
```

- Using Jython:

```
AdminTask.listKeyManagers()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listKeyManagers {-interactive}
```

- Using Jython:

```
AdminTask.listKeyManagers('-interactive')
```

modifyKeyManager

The **modifyKeyManager** command changes existing key manager settings.

Target object

None.

Required parameters

-name
The name that uniquely identifies the key manager. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-provider
Specifies the provider name of the key manager. (String, optional)

-algorithm
Specifies the algorithm name of the key manager. (String, optional)

-keyManagerClass
Specifies the name of the key manager implementation class. You cannot use this parameter with the provider or the algorithm parameter. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyKeyManager {-name testKM -provider IBMJSSE2 -algorithm IbmX509}
```

- Using Jython string:

```
AdminTask.modifyKeyManager('[-name testKM -provider IBMJSSE2 -algorithm IbmX509]')
```

- Using Jython list:

```
AdminTask.modifyKeyManager(['-name', 'testKM', '-provider', 'IBMJSSE2', '-algorithm', 'IbmX509'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyKeyManager {-interactive}
```

- Using Jython:

```
AdminTask.modifyKeyManager('-interactive')
```

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

"Creating an SSL configuration at the node scope using scripting" on page 631
An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

KeyStoreCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure keystores with the wsadmin tool. A keystore is created by the application server during install and can contain cryptographic keys or certificates. The commands and parameters in the KeyStoreCommands group can be used to create, delete, and manage keystores.

The KeyStoreCommands command group for the AdminTask object includes the following commands:

• “modifyKeyStore” on page 662

changeKeyStorePassword

The **changeKeyStorePassword** command modifies the password of a keystore. The command automatically saves the new password to the configuration.

Required parameters

-keyStoreName
Specifies the name of the password to change. (String, required)

-keyStorePassword
Specifies the name of the password to change. (String, required)

-newKeyStorePassword
Specifies the new password that to use to access the keystore. (String, required)

-newKeyStorePasswordVerify
Specifies the new password to confirm the new keystore password. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the keystore. (String, optional)

Examples

Batch mode example usage:

• Using Jacl:

```
$AdminTask changeKeyStorePassword {-keystoreName myKeystore -keyStorePassword
WebAS -newKeyStorePassword newpwd -newKeyStorePasswordVerify newpwd}
```

• Using Jython string:

```
AdminTask.changeKeyStorePassword('[-keystoreName myKeystore -keyStorePassword
WebAS -newKeyStorePassword newpwd -newKeyStorePasswordVerify newpwd]')
```

• Using Jython list:

```
AdminTask.changeKeyStorePassword(['-keystoreName', 'myKeystore', '-keyStorePassword',
'WebAS', '-newKeyStorePassword', 'newpwd', '-newKeyStorePasswordVerify', 'newpwd'])
```

Interactive mode example usage:

• Using Jacl:

```
$AdminTask changeKeyStorePassword {-interactive}
```

• Using Jython:

```
AdminTask.changeKeyStorePassword('-interactive')
```

changeMultipleKeyStorePasswords

The **changeMultipleKeyStorePasswords** command updates the passwords for each keystores in the configuration that has a specific password. This is useful because when you create keystore files on the system, they will have `WebAS` as a password by default.

Required parameters

-keyStorePassword
Specifies the name of the password that you want to change. (String, required)

-newKeyStorePassword
Specifies the new password that you will use to access the keystore. (String, required)

-newKeyStorePasswordVerify
Confirms the new keystore password. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask changeMultipleKeyStorePasswords {-keyStorePassword WebAS
-newKeyStorePassword newpwd -newKeyStorePasswordVerify newpwd}
```

- Using Jython string:

```
AdminTask.changeMultipleKeyStorePasswords('[-keyStorePassword WebAS
-newKeyStorePassword newpwd -newKeyStorePasswordVerify newpwd]')
```

- Using Jython list:

```
AdminTask.changeMultipleKeyStorePasswords(['-keyStorePassword', 'WebAS',
'-newKeyStorePassword', 'newpwd', '-newKeyStorePasswordVerify', 'newpwd'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask changeMultipleKeyStorePasswords {-interactive}
```

- Using Jython:

```
AdminTask.changeMultipleKeyStorePasswords('-interactive')
```

createKeyStore

The **createKeyStore** command creates the keystore settings in the configuration and the keystore database.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-keyStoreType
The implementation of the keystore management. (String, required)

-keyStoreLocation
The location of the keystore. For file based, the location is the files system path to the keystore database. For hardware keystore, the location is the path to the token library. (String, required)

If you create the IBMi5OSKeyStore keystore, the keystore location must include the `.kdb` file extension.

-keyStorePassword
The password that protects the keystore. (String, required)

-keyStorePasswordVerify
The password that protects the keystore. (String, required)

Optional parameters

-keyStoreProvider
The provider used to implement the keystore. (String, optional)

-keyStoreIsFileBased
Set the value of this parameter to `true` if the keystore is file based. Set the value of this parameter to `false` for hardware crypto keystores. (Boolean, optional)

-keyStoreHostList
A list of host names that indicate from where the keystore is remotely managed, separated by commas. (String, optional)

-keyStoreInitAtStartup
Set the value of this parameter to `true` if the keystore is initialized at startup. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-keyStoreReadOnly
Set the value of this parameter to `true` if you cannot write to the keystore. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-keyStoreStashFile
Set the value of this parameter to `true` if you want to create stash files for CMS type keystore. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-enableCryptoOperations
Specifies if the keystore object will be used for hardware cryptographic operations or not. The default value is `false`. (Boolean, optional)

-keyStoreDescription
Specifies user defined text to describe the keystore of interest. (String, optional)

-keyStoreUsage
Specifies the keystore usage of interest. Specify `SSLKeys`, `KeySetKeys`, `RootKeys`, `DeletedKeys`, `DefaultSigners`, or `RSATokenKeys`. (String, optional)

-scopeName
The name that uniquely identifies the management scope, for example: `(cell):localhostNode01Cell`. (String, optional)

-controlRegionUser
Specifies the control region user to create a writable keystore object for the control regions key ring. Specify this option for SAF key rings when SAF writable key rings is enabled. (String, optional)

-servantRegionUser
Specifies the servant region user to create a writable keystore object for the servant regions key ring. Specify this option for SAF key rings when SAF writable key rings is enabled. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createKeyStore {-keyStoreName testKS -keyStoreType JCEKS
-keyStoreLocation c:\temp\testKeyFile.p12 -keyStorePassword testpwd
-keyStorePasswordVerify testpwd -keyStoreIsFileBased true -keyStoreInitAtStartup
true -keyStoreReadOnly false}
```

- Using Jython string:

```
AdminTask.createKeyStore('[-keyStoreName testKS -keyStoreType JCEKS -keyStoreLocation
c:\temp\testKeyFile.p12 -keyStorePassword testpwd -keyStorePasswordVerify testpwd
-keyStoreIsFileBased true -keyStoreInitAtStartup true -keyStoreReadOnly false]')
```

- Using Jython list:

```
AdminTask.createKeyStore(['-keyStoreName', 'testKS', '-keyStoreLocation', '-keyStoreType',
'JCEKS', 'c:\temp\testKeyFile.p12', '-keyStorePassword', 'testpwd',
'-keyStorePasswordVerify', 'testpwd', '-keyStoreIsFileBased', 'true',
'-keyStoreInitAtStartup', 'true', '-keyStoreReadOnly', 'false'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createKeyStore {-interactive}
```

- Using Jython:

```
AdminTask.createKeyStore('-interactive')
```

createCMSKeyStore

The **createCMSKeyStore** command creates a CMS keystore database and the keystore settings in the configuration.

Required parameters

-cmsKeyStoreURI
The URI of the CMS keystore. (String, required)

-pluginHostName
The host name of the plug-in. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createCMSKeyStore {-cmsKeyStoreURI CMSKeystoreURI -pluginHostName myHostName}
```

- Using Jython string:

```
AdminTask.createCMSKeyStore('-cmsKeyStoreURI CMSKeystoreURI -pluginHostName myHostName')
```

- Using Jython list:

```
AdminTask.createCMSKeyStore(['-cmsKeyStoreURI', 'CMSKeystoreURI', '-pluginHostName',
'myHostName'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCMSKeyStore {-interactive}
```

- Using Jython:

```
AdminTask.createCMSKeyStore('-interactive')
```

deleteKeyStore

The **deleteKeyStore** command deletes the settings of a keystore from the configuration and the keystore file.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore that you want to delete. (String, required)

Optional parameters

-scopeName
The name that uniquely identifies the management scope, for example: `(cell):localhostNode01Cell`. (String, optional)

-removeKeyStoreFile
Specifies whether to remove the keystore file. Specify `true` to remove the keystore file or `false` to keep the keystore file in your configuration. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteKeyStore {-keyStoreName testKS}
```

- Using Jython string:

```
AdminTask.deleteKeyStore('[-keyStoreName testKS]')
```

- Using Jython list:

```
AdminTask.deleteKeyStore(['-keyStoreName', 'testKS'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteKeyStore {-interactive}
```

- Using Jython:

```
AdminTask.deleteKeyStore('-interactive')
```

exchangeSigners

The **exchangeSigners** command exchange signer certificate between keystores.

Required parameters

-keyStoreName1
The name that uniquely identifies a keystore. You must specify a second keystore name using the keyStoreName2 parameter. (String, required)

-keyStoreScope1
The scope name of the keystore that you specified with the keyStoreName1 parameter. (String, required)

-keyStoreName2
The name that uniquely identifies a keystore. You must specify a second keystore name using the keyStoreName1 parameter. (String, required)

-keyStoreScope2
The scope name of the keystore that you specified with the keyStoreName2 parameter. (String, required)

Optional parameters

-certificateAlaisList1
A list of aliases separated by a comma. (String, optional)

-certificateAliasList2
A list of aliases separated by a comma. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask exchangeSigners {-keyStoreName1 testKS -certificateAliasList1 testCert1
-keyStoreName2 secondKS -certificateAlaisList2 certAlis}
```

- Using Jython string:

```
AdminTask.exchangeSigners('[-keyStoreName1 testKS -certificateAliasList1 testCert1
-keyStoreName2 secondKS -certificateAlaisList2 certAlis]')
```

- Using Jython list:

```
AdminTask.exchangeSigners(['-keyStoreName1', 'testKS', '-certificateAliasList1',
'testCert1', '-keyStoreName2', 'secondKS', '-certificateAlaisList2',
'certAlis'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exchangeSigners {-interactive}
```

- Using Jython:

```
AdminTask.exchangeSigners('-interactive')
```

getKeyStoreInfo

The **getKeyStoreInfo** command displays the settings of a particular keystore.

Required parameters

-name
The name that uniquely identifies the keystore. (String, required)

Optional parameters

-scopeName
The name that uniquely identifies the management scope, for example: `(cell):localhostNode01Cell`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getKeyStoreInfo {-name testKS}
```

- Using Jython string:

```
AdminTask.getKeyStoreInfo('[-name testKS]')
```

- Using Jython list:

```
AdminTask.getKeyStoreInfo(['-name', 'testKS'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getKeyStoreInfo {-interactive}
```

- Using Jython:

```
AdminTask.getKeyStoreInfo('-interactive')
```

listKeyFileAliases

The **listKeyFileAliases** command lists the certificates in a keystore file.

Required parameters

-keyFilePath
The path of the key file. (String, required)

-keyFilePassword
The password for the key file. (String, required)

-keyFileType
The key file type. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listKeyFileAliases {-keyFilePath /temp/testKeyFile.p12
-keyFilePassword testPwd -keyFileType PKCS12}
```

- Using Jython string:

```
AdminTask.listKeyFileAliases('[-keyFilePaht /temp/testKeyFile.p12
-keyFilePassword testPwd -keyFileType PKCS12]')
```

- Using Jython list:

```
AdminTask.listKeyFileAliases(['-keyFilePaht', '/temp/testKeyFile.p12',
'-keyFilePassword', 'testPwd', '-keyFileType', 'PKCS12'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listKeyFileAliases {-interactive}
```

- Using Jython:

```
AdminTask.listKeyFileAliases('-interactive')
```

listKeyStores

The **listKeyStores** command lists the keystore for a particular scope.

Required parameters

None.

Optional parameters

-displayObjectName
Specify the value of this parameter as `true` to list the keystore configuration objects within a scope. Set the value of this parameter to `false` to list the strings that contain the keystore name and management scope. (Boolean, optional)

-scopeName
Specifies the name that uniquely identifies the management scope, for example: `(cell):localhostNode01Cell`. (String, optional)

-all
Specify the value of this parameter as `true` to list all keystores. This parameter overrides the scopeName parameter. The default value is `false`. (Boolean, optional)

-keyStoreUsage
Specifies the keystore usage of interest. Specify `SSLKeys`, `KeySetKeys`, `RootKeys`, `DeletedKeys`, `DefaultSigners`, or `RSATokenKeys`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listKeyStores
```

- Using Jython:

```
AdminTask.listKeyStores()
```

Interactive mode example usage:

listKeyStoreTypes

The **listKeyStoreTypes** command lists all valid keystore types.

Required parameters

None.

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listKeyStoreTypes
```

- Using Jython:

```
AdminTask.listKeyStoreTypes()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listKeyStoreTypes {-interactive}
```

- Using Jython string:

```
AdminTask.listKeyStoreTypes('-interactive')
```

modifyKeyStore

The **modifyKeyStore** command modifies attributes for an existing keystore. Only some keystore attributes are modifiable, depending on what you are modifying. Use the following guidelines to use the command:

- To use this command to change the keystore file that the keystore object references, specify the keyStoreName, keyStoreLocation, keyStoreType, and keyStorePassword parameters.
-

Required parameters

-keyStoreName
Specifies the unique name that identifies the keystore. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the keystore. (String, optional)

-keyStoreType
Specifies one of the predefined keystore types. Valid values are `JCEKS`, `CMSKS`, `PKCS12`, `PKCS11`, and `JKS`. (String, optional)

-keyStoreLocation
Specifies the fully qualified location of the keystore file. To modify the location of the keystore file, you must specify the keyStoreLocation, keyStoreType, keyStorePassword, and keyStoreName parameters. (String, optional)

-keyStorePassword
Specifies the password to open the keystore. Use the changeKeystorePassword command to change the password of the keystore. (String, optional)

-keyStoreIsFileBased
Specifies whether the keystore is file based. To modify whether the keystore is file-based, specify the keyStoreIsFileBased and keyStoreName parameters. (Boolean, optional)

-keyStoreInitAtStartup
Specifies whether the keystore initiates at server startup. To modify whether the keystore initiates at server startup, specify the keyStoreInitAtStartup and keyStoreName parameters. (Boolean, optional)

-keyStoreReadOnly
Specifies whether the keystore is writable. To modify whether the keystore is read-only, specify the keyStoreReadOnly and keyStoreName parameters. (Boolean, optional)

-keyStoreDescription
Specifies a statement that describes the keystore. To modify the keystore description, specify the keyStoreDescription and keyStoreName parameters. (String, optional)

-keyStoreUsage
Specifies the keystore usage of interest. Specify `SSLKeys`, `KeySetKeys`, `RootKeys`, `DeletedKeys`, `DefaultSigners`, or `RSATokenKeys`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyKeyStore {-keyStoreName CellDefaultKeyStore
-keyStoreLocation c:\temp\testKeyFile.p12 -keyStoreType JCEKS
-keyStorePassword my1password}
```

- Using Jython:

```
AdminTask.modifyKeyStore('-keyStoreName CellDefaultKeyStore -keyStoreLocation
c:\temp\testKeyFile.p12 -keyStoreType JCEKS -keyStorePassword my1password')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyKeyStore {-interactive}
```

- Using Jython:

```
AdminTask.modifyKeyStore('-interactive')
```

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

"Creating an SSL configuration at the node scope using scripting" on page 631
An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

SSLConfigCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the SSLConfigCommands group can be used to create and manage Secure Sockets Layer (SSL) configurations and properties.

The SSLConfigCommands command group for the AdminTask object includes the following commands:

createSSLConfig

The createSSLConfig command creates an SSL configuration that is based on key store and trust store settings. You can use the SSL configuration settings to make the SSL connections.

Target object

None.

Required parameters

-alias
The name of the alias. (String, required)

-trustStoreNames
The key store that holds trust information used to validate the trust from remote connections. (String, required)

-keyStoreName
The key store that holds the personal certificates that provide identity for the connection. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

-clientKeyAlias
The certificate alias name for the client. (String, optional)

-serverKeyAlias
The certificate alias name for the server. (String, optional)

-type
The type of SSL configuration. (String, optional)

-clientAuthentication
Set the value of this parameter to `true` to request client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-securityLevel
The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, optional)

-enabledCiphers
A list of ciphers used during SSL handshake. (String, optional)

-jsseProvider
One of the JSSE providers. (String, optional)

-clientAuthenticationSupported
Set the value of this parameter to `true` to support client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-sslProtocol
The protocol type for the SSL handshake. Valid values include: `SSL_TLS`, `SSL`, `SSLv2`, `SSLv3`, `TLS`, `TLSv1`. (String, optional)

-trustManagerObjectName
A list of trust managers separated by commas. (String, optional)

-trustStoreScopeName
The management scope name of the trust store. (String, optional)

-keyStoreScopeName
The management scope name of the key store. (String, optional)

-ssslKeyRingName
Specifies a system SSL (SSSL) key ring name. The value for this parameter has no affect unless the SSL configuration type is SSSL. (String, optional)

Example output

The command returns the configuration object name of the new SSL configuration object.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createSSLConfig {-alias testSSLCfg  -clientKeyAlias key1
-serverKeyAlias key2 -trustStoreNames trustKS –keyStoreName
testKS -keyManagerName testKeyMgr}
```

- Using Jython string:

```
AdminTask.createSSLConfig('[-alias testSSLCfg  -clientKeyAlias key1
-serverKeyAlias key2 -trustStoreNames trustKS –keyStoreName
testKS -keyManagerName testKeyMgr]')
```

- Using Jython list:

```
AdminTask.createSSLConfig(['-alias', 'testSSLCfg', '-clientKeyAlias',
'key1', '-serverKeyAlias', 'key2', '-trustStoreNames', 'trustKS',
'–keyStoreName', 'testKS', '-keyManagerName', 'testKeyMgr'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createSSLConfig {-interactive}
```

- Using Jython:

```
AdminTask.createSSLConfig('-interactive')
```

createSSLConfigProperty

The createSSLConfigProperty command creates a property for an SSL configuration. Use this command to set SSL configuration settings that are different than the settings in the SSL configuration object.

Target object

None.

Required parameters

-sslConfigAliasName
The alias name of the SSL configuration. (String, required)

-propertyName
The name of the property. (String, required)

-propertyValue
The value of the property. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createSSLConfigProperty {-sslConfigAliasName NodeDefaultSSLSettings
-scopeName (cell):localhostNode01Cell:(node):localhostNode01 -propertyName
test.property -propertyValue testValue}
```

- Using Jython string:

```
AdminTask.createSSLConfigProperty('[-sslConfigAliasName NodeDefaultSSLSettings
-scopeName (cell):localhostNode01Cell:(node):localhostNode01 -propertyName
test.property -propertyValue testValue]')
```

- Using Jython list:

```
AdminTask.createSSLConfigProperty(['-sslConfigAliasName', 'NodeDefaultSSLSettings',
'-scopeName', '(cell):localhostNode01Cell:(node):localhostNode01', '-propertyName',
'test.property', '-propertyValue', 'testValue'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createSSLConfigProperty {-interactive}
```

- Using Jython:

```
AdminTask.createSSLConfigProperty('-interactive')
```

deleteSSLConfig

The deleteSSLConfig command deletes the SSL configuration object that you specify from the configuration.

Target object

None.

Required parameters and return values

-alias
The name of the alias. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteSSLConfig {-alias NodeDefaultSSLSettings -scopeName
(cell):localhostNode01Cell:(node):localhostNode01}
```

- Using Jython string:

```
AdminTask.deleteSSLConfig('[-alias NodeDefaultSSLSettings -scopeName
(cell):localhostNode01Cell:(node):localhostNode01]')
```

- Using Jython list:

```
AdminTask.deleteSSLConfig(['-alias', 'NodeDefaultSSLSettings', '-scopeName',
'(cell):localhostNode01Cell:(node):localhostNode01'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteSSLConfig {-interactive}
```

- Using Jython:

```
AdminTask.deleteSSLConfig('-interactive')
```

getSSLConfig

The getSSLConfig command obtains information about an SSL configuration and displays the settings.

Target object

None.

Required parameters and return values

-alias
The name of the alias. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

Example output

The command returns information about the SSL configuration of interest.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfig {-alias NodeDefaultSSLSettings -scopeName
(cell):localhostNode01Cell:(node):localhostNode01}
```

- Using Jython string:

```
AdminTask.getSSLConfig('[-alias NodeDefaultSSLSettings -scopeName
(cell):localhostNode01Cell:(node):localhostNode01]')
```

- Using Jython list:

```
AdminTask.getSSLConfig(['-alias', 'NodeDefaultSSLSettings', '-scopeName',
'(cell):localhostNode01Cell:(node):localhostNode01'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfig {-interactive}
```

- Using Jython:

```
AdminTask.getSSLConfig('-interactive')
```

getSSLConfigProperties

The getSSLConfigProperties command obtains information about SSL configuration properties.

Target object

None.

Required parameters and return values

-alias
The name of the alias. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

Example output

The command returns additional information about the SSL configuration properties.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfigProperties {-sslConfigAliasName NodeDefaultSSLSettings
-scopeName (cell):localhostNode01Cell:(node):localhostNode01}
```

- Using Jython string:

```
AdminTask.getSSLConfigProperties('[-sslConfigAliasName NodeDefaultSSLSettings
-scopeName (cell):localhostNode01Cell:(node):localhostNode01]')
```

- Using Jython list:

```
AdminTask.getSSLConfigProperties(['-sslConfigAliasName', 'NodeDefaultSSLSettings',
'-scopeName', '(cell):localhostNode01Cell:(node):localhostNode01'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfigProperties {-interactive}
```

- Using Jython:

```
AdminTask.getSSLConfigProperties('-interactive')
```

listSSLCiphers

The listSSLCiphers command lists the SSL ciphers.

Target object

None.

Required parameters

-sslConfigAliasName
The alias name of the SSL configuration. (String, required)

-securityLevel
The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

-provider
(String, optional)

Example output

The command returns a list of SSL ciphers.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listSSLCiphers {-sslConfigAliasName testSSLCfg
-securityLevel HIGH}
```

- Using Jython string:

```
AdminTask.listSSLCiphers('[-sslConfigAliasName testSSLCfg
-securityLevel HIGH]')
```

- Using Jython list:

```
AdminTask.listSSLCiphers(['-sslConfigAliasName', 'testSSLCfg',
'-securityLevel', 'HIGH'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSSLCiphers {-interactive}
```

- Using Jython:

```
AdminTask.listSSLCiphers('-interactive')
```

listSSLConfigs

The listSSLConfigs command lists the defined SSL configurations within a management scope.

Target object

None.

Optional parameters

-scopeName
The name of the scope. (String, optional)

-displayObjectName
Set the value of this parameter to `true` to list the SSL configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the SSL configuration alias and management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all SSL configurations. This parameter overrides the scopeName parameter. The default value is `false`. (Boolean, optional)

Example output

The command returns a list of defined SSL configurations.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigs {-scopeName (cell): localhostNode01Cell:(node):localhostNode01
-displayObjectName true}
```

- Using Jython string:

```
AdminTask.listSSLConfigs('[-scopeName (cell):localhostNode01Cell:(node):localhostNode01
-displayObjectName true]')
```

- Using Jython list:

```
AdminTask.listSSLConfigs(['-scopeName', '(cell):localhostNode01Cell:(node):localhostNode01',
'-displayObjectName', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigs {-interactive}
```

- Using Jython:

```
AdminTask.listSSLConfigs('-interactive')
```

listSSLConfigProperties

The listSSLConfigProperties command lists the properties for a SSL configuration.

Target object

None.

Required parameters

-alias
The alias name of the SSL configuration. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

-displayObjectName
Set the value of this parameter to `true` to list the SSL configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the SSL configuration alias and management scope. (Boolean, optional)

Example output

The command returns SSL configuration properties.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigProperty {-alias SSL123 -scopeName
(cell):localhostNode01Cell:(node):localhostNode01 -displayObjectName true}
```

- Using Jython string:

```
AdminTask.listSSLConfigProperty('[-alias SSL123 -scopeName
(cell):localhostNode01Cell:(node):localhostNode01 -displayObjectName true]')
```

- Using Jython list:

```
AdminTask.listSSLConfigProperty(['-alias', 'SSL123', '-scopeName',
'(cell):localhostNode01Cell:(node):localhostNode01', '-displayObjectName', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigProperties {-interactive}
```

- Using Jython:

```
AdminTask.listSSLConfigProperties('-interactive')
```

listSSLRepertoires

The listSSLRepertoires command lists all of the Secure Sockets Layer (SSL) configuration instances that you can associate with an SSL inbound channel.If you create a new SSL alias using the administrative console, the alias name is automatically created in the *node_name/alias_name* format. However, if you create a new SSL alias using the wsadmin tool, you must create the SSL alias and specify both the node name and alias name in the *node_name/alias_name* format.

Target object

SSLInboundChannel instance for which the SSLConfig candidates are listed.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of eligible SSL configuration object names.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listSSLRepertoires SSL_3(cells/mybuildCell01/nodes/mybuildNode01/servers/
server2|server.xml#SSLInboundChannel_1093445762330)
```

- Using Jython string:

```
print AdminTask.listSSLRepertoires('SSL_3(cells/mybuildCell01/nodes/mybuildNode01/
servers/server2|server.xml#SSLInboundChannel_1093445762330)')
```

- Using Jython list:

```
print AdminTask.listSSLRepertoires('SSL_3(cells/mybuildCell01/nodes/mybuildNode01/
servers/server2|server.xml#SSLInboundChannel_1093445762330)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSSLRepertoires {-interactive}
```

- Using Jython:

```
print AdminTask.listSSLRepertoires('-interactive')
```

modifySSLConfig

The modifySSLConfig command modifies the settings of an existing SSL configuration.

Target object

None.

Required parameters

-alias
The name of the alias. (String, required)

Optional parameters

-scopeName
The name of the scope. (String, optional)

-clientKeyAlias
The certificate alias name for the client. (String, optional)

-serverKeyAlias
The certificate alias name for the server. (String, optional)

-type
The type of SSL configuration. (String, optional)

-clientAuthentication
Set the value of this parameter to `true` to request client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-securityLevel
The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, optional)

-enabledCiphers
A list of ciphers used during SSL handshake. (String, optional)

-jsseProvider
One of the JSSE providers. (String, optional)

-clientAuthenticationSupported
Set the value of this parameter to `true` to support client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-sslProtocol
The protocol type for the SSL handshake. Valid values include: `SSL_TLS`, `SSL`, `SSLv2`, `SSLv3`, `TLS`, `TLSv1`. (String, optional)

-trustManagerObjectNames
A list of trust managers separated by commas. (String, optional)

-trustStoreNames
The key store that holds trust information used to validate the trust from remote connections. (String, optional)

-trustStoreScopeName
The management scope name of the trust store. (String, optional)

-keyStoreName
The key store that holds the personal certificates that provide identity for the connection. (String, optional)

-keyStoreScopeName
The management scope name of the key store. (String, optional)

-ssslKeyRingName
Specifies a system SSL (SSSL) key ring name. The value for this parameter has no affect unless the SSL configuration type is SSSL. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifySSLConfig {-alias testSSLCfg -clientKeyAlias tstKey1
-serverKeyAlias tstKey2 -securityLevel LOW}
```

- Using Jython string:

```
AdminTask.modifySSLConfig('[-alias testSSLCfg -clientKeyAlias tstKey1
-serverKeyAlias tstKey2 -securityLevel LOW]')
```

- Using Jython list:

```
AdminTask.modifySSLConfig(['-alias', 'testSSLCfg', '-clientKeyAlias', 'tstKey1',
'-serverKeyAlias', 'tstKey2', '-securityLevel', 'LOW'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifySSLConfig {-interactive}
```

- Using Jython:

```
AdminTask.modifySSLConfig('-interactive')
```

SSLConfigGroupCommands group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the SSLConfigGroupCommands group can be used to create and manage SSL configuration groups.

The SSLConfigGroupCommands command group for the AdminTask object includes the following commands:

- “deleteSSLConfigGroup” on page 674

deleteSSLConfigGroup

The deleteSSLConfigGroup command deletes a SSL configuration group from the configuration.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the SSL configuration group. (String, required)

-direction
Specifies the direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required)

Optional parameters

-scopeName
Specifies the name that uniquely identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteSSLConfigGroup {-name createSSLCfgGrp -direction inbound}
```

- Using Jython string:

```
AdminTask.deleteSSLConfigGroup('[-name createSSLCfgGrp -direction inbound)']
```

- Using Jython list:

```
AdminTask.deleteSSLConfigGroup(['-name', 'createSSLCfgGrp', '-direction', 'inbound')]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteSSLConfigGroup {-interactive}
```

- Using Jython:

```
AdminTask.deleteSSLConfigGroup('-interactive')
```

getSSLConfigGroup

The getSSLConfigGroup command returns information about a SSL configuration setting.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the SSL configuration group. (String, required)

-direction
Specifies the direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required)

Optional parameters

-scopeName
Specifies the name that uniquely identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfigGroup {-name createSSLCfgGrp -direction inbound}
```

- Using Jython string:

```
AdminTask.getSSLConfigGroup('[-name createSSLCfgGrp -direction inbound)']
```

- Using Jython list:

```
AdminTask.getSSLConfigGroup(['-name', 'createSSLCfgGrp', '-direction', 'inbound')]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getSSLConfigGroup {-interactive}
```

- Using Jython:

```
AdminTask.getSSLConfigGroup('-interactive')
```

listSSLConfigGroups

The listSSLConfigGroups command lists the SSL configuration groups within a scope and a direction.

Target object

None.

Required parameters

None.

Optional parameters

-direction
Specifies the direction to which the SSL configuration applies. Valid values include `inbound` or `outbound`. (String, optional)

-scopeName
Specifies the name that uniquely identifies the management scope. (String, optional)

-displayObjectName
If you set this parameter to true, the command returns a list of all of the SSL configuration group objects within the scope. If you set this parameter to `false`, the command returns a list of strings that contain the SSL configuration name and management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all SSL configuration groups. This parameter overrides the scopeName parameter. The default value is `false`. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigGroups {-displayObjectName true}
```

- Using Jython string:

```
AdminTask.listSSLConfigGroups('[-displayObjectName true)']
```

- Using Jython list:

```
AdminTask.listSSLConfigGroups(['-displayObjectName' 'true')]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSSLConfigGroups {-interactive}
```

- Using Jython:

```
AdminTask.listSSLConfigGroups('-interactive')
```

modifySSLConfigGroup

The modifySSLConfigGroup command modifies the setting of an existing SSL configuration group.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the SSL configuration group. (String, required)

-direction
Specifies the direction to which the SSL configuration applies. Valid values include `inbound` or `outbound`. (String, required)

Optional parameters

-certificateAlias
Specifies a unique name to identify a certificate. (String, optional)

-scopeName
Specifies the name that uniquely identifies the management scope. (String, optional)

-sslConfigAliasName
Specifies the alias that uniquely identifies the SSL configurations in the group. (String, optional)

-sslConfigScopeName
Specifies the scope that uniquely identifies the SSL configurations in the group. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifySSLConfigGroup {-name createSSLCfgGrp -direction inbound -certificateAlias alias2}
```

- Using Jython string:

```
AdminTask.modifySSLConfigGroup('[-name createSSLCfgGrp -direction inbound -certificateAlias alias2)']
```

- Using Jython list:

```
AdminTask.modifySSLConfigGroup(['-name', 'createSSLCfgGrp', '-direction', 'inbound', '-certificateAlias', 'alias2')]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifySSLConfigGroup {-interactive}
```

- Using Jython:

```
AdminTask.modifySSLConfigGroup('-interactive')
```

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

"Creating an SSL configuration at the node scope using scripting" on page 631
An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

TrustManagerCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the TrustManagerCommands group can be used to create, delete, and query trust manager settings in your configuration. You can also use these commands to create a custom trust manager for a pure client.

The TrustManagerCommands command group for the AdminTask object includes the following commands:

- "deleteTrustManager"
- "getTrustManager" on page 678
- "listTrustManagers" on page 679
- "modifyTrustManager" on page 679

deleteTrustManager

The **deleteTrustManager** command deletes the trust manager settings from the configuration.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the trust manager. (String, required)

Optional parameters

-scopeName
Specifies the name of the scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteTrustManager {-name testTM}
```

- Using Jython string:

```
AdminTask.deleteTrustManager('[-name testTM]')
```

- Using Jython list:

```
AdminTask.deleteTrustManager(['-name', 'testTM'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteTrustManager {-interactive}
```

- Using Jython:

```
AdminTask.deleteTrustManager('-interactive')
```

getTrustManager

The **getTrustManager** command obtains the setting of a trust manager.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the trust manager. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getTrustManager {-name testTM}
```

- Using Jython string:

```
AdminTask.getTrustManager('[-name testTM]')
```

- Using Jython list:

```
AdminTask.getTrustManager(['-name', 'testTM'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getTrustManager {-interactive}
```

- Using Jython:

```
AdminTask.getTrustManager('-interactive')
```

listTrustManagers

The **listTrustManagers** command lists the trust managers within a particular management scope.

Target object

None.

Required parameters

None.

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-displayObjectName
Set the value of this parameter to true to list the trust manager objects within a scope. Set the value of this parameter to false to list the strings that contain the trust manager name and management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all trust managers. This parameter overrides the scopeName parameter. The default value is false. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listTrustManagers {-displayObjectName true}
  ```
- Using Jython string:
  ```
  AdminTask.listTrustManagers('[-displayObjectName true]')
  ```
- Using Jython list:
  ```
  AdminTask.listTrustManagers(['-displayObjectName', 'true'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listTrustManagers {-interactive}
  ```
- Using Jython:
  ```
  AdminTask.listTrustManagers('-interactive')
  ```

modifyTrustManager

The **modifyTrustManager** command changes existing trust manager settings.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the trust manager. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-provider
Specifies the provider name of the trust manager. (String, optional)

-algorithm
Specifies the algorithm name of the trust manager. (String, optional)

-trustManagerClass
Specifies a class that implements the `javax.net. sslX509Tr ustManager` interface. You cannot use this parameter with the provider or algorithm parameters. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyTrustManager {-name testTM -trustManagerClass test.trust.manager}
```

- Using Jython string:

```
AdminTask.modifyTrustManager('[-name testTM -trustManagerClass test.trust.manager]')
```

- Using Jython list:

```
AdminTask.modifyTrustManager(['-name', 'testTM', '-trustManagerClass', 'test.trust.manager'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyTrustManager {-interactive}
```

- Using Jython:

```
AdminTask.modifyTrustManager('-interactive')
```

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

Creating a custom trust manager configuration
You can create a custom trust manager configuration at any management scope and associate the new trust manager with a Secure Sockets Layer (SSL) configuration.

KeySetCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the KeySetCommands group can be used to create, delete, and query for key set settings in your configuration.

The KeySetCommands command group for the AdminTask object includes the following commands:

createKeySet

The **createKeySet** command creates the key set settings in the configuration. Use this command to control key instances that have the same type.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set. (String, required)

-aliasPrefix
Specifies the prefix for the key alias when a new key generates. (String, required)

-password
Specifies the password that protects the key in the keystore. (String, required)

-maxKeyReferences
Specifies the maximum number of key references from the returned keys in the key set of interest. (Integer, required)

-keyStoreName
Specifies the keystore that contains the keys. (String, required)

Optional parameters

-scopeName
Specifies the unique name of the management scope. (String, optional)

-deleteOldKeys
Set the value of this parameter to `true` to delete old keys when new keys are generated. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-keyGenerationClass
Specifies the class that is used to generate new keys in the key set. (String, optional)

-keyStoreScopeName
Specifies the management scope where the keystore is located. (String, optional)

-isKeyPair
Set the value of this parameter to `true` if the keys in the key set are key pairs. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

Example output

The command returns the configuration object name of the key set object that you created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createKeySet {-name testKeySet –aliasPrefix test –password pwd
–maxKeyReferences 2 –deleteOldKeys true –keyStoreName testKeyStore –isKeyPair false}
```

- Using Jython string:

```
AdminTask.createKeySet('[-name testKeySet –aliasPrefix test –password pwd
–maxKeyReferences 2 –deleteOldKeys true –keyStoreName testKeyStore –isKeyPair false]')
```

- Using Jython list:

```
AdminTask.createKeySet(['-name', 'testKeySet', '–aliasPrefix', 'test',
'–password', 'pwd', '–maxKeyReferences', '2', '–deleteOldKeys', 'true',
'–keyStoreName', 'testKeyStore', '–isKeyPair', 'false'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createKeySet {-interactive}
```

- Using Jython string:

```
AdminTask.createKeySet ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createKeySet (['-interactive'])
```

deleteKeySet

The **deleteKeySet** command deletes the settings of a key set from the configuration.

Target object

None.

Required parameters

-name
The name that uniquely identifies the key set. (String, required)

Optional parameters

-scopeName
Specifies the unique name of the management scope. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteKeySet{ -name testKeySet}
```

- Using Jython string:

```
AdminTask.deleteKeySet('[-name testKeySet]')
```

- Using Jython list:

```
AdminTask.deleteKeySet(['-name', 'testKeySet'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteKeySet {-interactive}
```

- Using Jython string:

```
AdminTask.deleteKeySet ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteKeySet (['-interactive'])
```

generateKeyForKeySet

The **generateKeyForKeySet** command generates keys for the keys in the key set.

Target object

None.

Required parameters

-keySetName
Specifies the name of the key set. (String, required)

Optional parameters

-keySetScope
Specifies the scope of the key set. (String, optional)

-keySetSaveConfig
Set the value of this parameter to `true` to save the configuration of the key set. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask  generateKeyForKeySet{ -keySetName testKeySet }
```

- Using Jython string:

```
AdminTask.generateKeyForKeySet('[-keySetName testKeySet]')
```

- Using Jython list:

```
AdminTask.generateKeyForKeySet(['-keySetName', 'testKeySet'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask generateKeyForKeySet {-interactive}
```

- Using Jython string:

```
AdminTask.generateKeyForKeySet ('[-interactive]')
```

- Using Jython list:

```
AdminTask.generateKeyForKeySet (['-interactive'])
```

getKeySet

The **getKeySet** command displays the settings of a particular key set.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set. (String, required)

Optional parameters

-scopeName
Specifies the unique name of the management scope. (String, optional)

Example output

The command returns the settings of the specified key set group.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getKeySet {-name testKeySet}
  ```
- Using Jython string:
  ```
  AdminTask.getKeySet ('[-name testKeySet]')
  ```
- Using Jython list:
  ```
  AdminTask.getKeySet (['-name', 'testKeySet'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getKeySet {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getKeySet ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getKeySet (['-interactive'])
  ```

listKeySets

The **listKeySets** command lists the key sets in a particular scope.

Target object

None.

Required parameters

None.

Optional parameters

-scopeName
Specifies the unique name of the management scope. (String, optional)

-displayObjectNames
Set the value of this parameter to `true` to list the key set configuration objects within the scope. Set the value of this parameter to `false` if you want to list the strings that contain the key set group name and management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all key sets. This parameter overrides the scopeName parameter. The default value is `false`. (Boolean, optional)

Example output

The command returns the key sets for the scope that you specified.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listKeySets {-displayObjectName true}
  ```
- Using Jython string:
  ```
  AdminTask.listKeySets ('[-displayObjectName true]')
  ```
- Using Jython list:
  ```
  AdminTask.listKeySets (['-displayObjectName', 'true'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listKeySets {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listKeySets ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listKeySets (['-interactive'])
  ```

modifyKeySet

The **modifyKeySet** command changes the settings of an existing key set.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set. (String, required)

Optional parameters

-scopeName
Specifies the unique name of the management scope. (String, optional)

-aliasPrefix
Specifies the prefix for the key alias when a new key generates. (String, optional)

-password
Specifies the password that protects the key in the keystore. (String, optional)

-maxKeyReferences
Specifies the maximum number of key references from the returned keys in the key set of interest. (Integer, optional)

-deleteOldKeys
Set the value of this parameter to `true` to delete old keys when new keys are generated. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-keyGenerationClass
Specifies the class that is used to generate new keys in the key set. (String, optional)

-keyStoreName
Specifies the keystore that contains the keys. (String, optional)

-keyStoreScopeName
Specifies the management scope where the keystore is located. (String, optional)

-isKeyPair
Set the value of this parameter to `true` if the keys in the key set are key pairs. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyKeySet {-name testKeySet -maxKeyReferences 3
–deleteOldKeys false}
```

- Using Jython string:

```
AdminTask.modifyKeySet ('[-name testKeySet -maxKeyReferences 3
–deleteOldKeys false]')
```

- Using Jython list:

```
AdminTask.modifyKeySet (['-name', 'testKeySet', '-maxKeyReferences', '3',
'–deleteOldKeys', 'false'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyKeySet {-interactive}
```

- Using Jython string:

```
AdminTask.modifyKeySet ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifyKeySet (['-interactive'])
```

KeyReferenceCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the KeyReferenceCommands group can be used to create and manage the key reference settings for key set objects in your configuration.

The KeyReferenceCommands command group for the AdminTask object includes the following commands:

- “createKeyReference” on page 687

createKeyReference

The **createKeyReference** command creates the key reference setting in the configuration for key set objects.

Target object

None.

Required parameters and return values

-keySetName
The name that uniquely identifies the key set to which the key reference belongs. (String, required)

-keySetScope
The management scope of the key set. (String, optional)

-keyAlias
The alias name that identifies the key for the key set that you specify. (String, required)

-keyPassword
The password used for encrypting the key. (String, optional)

-keyPasswordVerify
The password used for encrypting the key. (String, optional)

-version
The version of the key reference. (String, optional)

-keyReferenceSaveConfig
Set the value of this parameter to `true` to save the key reference to the configuration. Otherwise, set the value to `false`. (String, optional)

- Returns: The configuration object name of the key reference scope object that you created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createKeyReference {-keySetName testKeySet  -keyAlias testKey
–password testPWD –passwordVerify testPWD –keyReferenceSaveConfig true}
```

- Using Jython string:

```
AdminTask.createKeyReference ('[-keySetName testKeySet  -keyAlias testKey
–password testPWD –passwordVerify testPWD –keyReferenceSaveConfig true]')
```

- Using Jython list:

```
AdminTask.createKeyReference (['-keySetName', 'testKeySet', '-keyAlias', 'testKey',
'–password', 'testPWD', '–passwordVerify', 'testPWD', '–keyReferenceSaveConfig', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createKeyReference {-interactive}
```

- Using Jython string:

```
AdminTask.createKeyReference ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createKeyReference (['-interactive'])
```

deleteKeyReference

The **deleteKeyReference** command deletes a key reference object from the key set object in the configuration.

Target object

None.

Required parameters and return values

-keySetName
The name that uniquely identifies the key set to which the key reference belongs. (String, required)

-keySetScope
The management scope of the key set. (String, optional)

-keyAlias
The alias name that identifies the key for the key set that you specify. (String, required)

- Returns: None.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask deleteKeyReference { -keySetName testKeySet –keyAlias testKey }
```
- Using Jython string:
```
AdminTask.deleteKeyReference ('[-keySetName testKeySet –keyAlias testKey]')
```
- Using Jython list:
```
AdminTask.deleteKeyReference (['-keySetName', 'testKeySet', '–keyAlias', 'testKey'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask deleteKeyReference {-interactive}
```
- Using Jython string:
```
AdminTask.deleteKeyReference ('[-interactive]')
```
- Using Jython list:
```
AdminTask.deleteKeyReference (['-interactive'])
```

getKeyReference

The **getKeyReference** command displays the setting of a key reference object.

Target object

None.

Required parameters and return values

-keySetName
The name that uniquely identifies the key set to which the key reference belongs. (String, required)

-keySetScope
The management scope of the key set. (String, optional)

-keyAlias
The alias name that identifies the key for the key set that you specify. (String, required)

- Returns: The settings of the key reference object.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getKeyReference { -keySetName testKeySet –keyAlias testKey }
```

- Using Jython string:

```
AdminTask.getKeyReference ('[-keySetName testKeySet –keyAlias testKey]')
```

- Using Jython list:

```
AdminTask.getKeyReference (['-keySetName', 'testKeySet', '–keyAlias', 'testKey'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getKeyReference {-interactive}
```

- Using Jython string:

```
AdminTask.getKeyReference ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getKeyReference (['-interactive'])
```

listKeyReferences

The **listKeyReferences** command lists the key references for a particular key set in the configuration.

Target object

None.

Required parameters and return values

-keySetName
The name that uniquely identifies the key set to which the key reference belongs. (String, required)

-keySetScope
The management scope of the key set. (String, optional)

- Returns: The configuration object name of the key reference scope object that you created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listKeyRefereces { -keySetName testKeySet}
```

- Using Jython string:

```
AdminTask.listKeyRefereces ('[-keySetName testKeySet]')
```

- Using Jython list:

```
AdminTask.listKeyRefereces (['-keySetName', 'testKeySet'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listKeyReferences {-interactive}
```

- Using Jython string:
  ```
  AdminTask.listKeyReferences ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listKeyReferences (['-interactive'])
  ```

KeySetGroupCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the KeySetGroupCommands group can be used to create and manage key set groups. Use these commands to manage groups of public, private, and shared keys.

The KeySetGroupCommands command group for the AdminTask object includes the following commands:

deleteKeySetGroup

The **deleteKeySetGroup** command deletes the settings of a key set group from the configuration.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set group. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

generateKeysForKeySetGroup

The **generateKeysForKeySetGroup** command generates keys for all of the keys in the key sets that make up the key set group.

Target object

None.

Required parameters

-keySetGroupName
Specifies the name of the key set group. (String, required)

Optional parameters

-keySetGroupScope
Specifies the scope of the key set group. (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

getKeySetGroup

The **getKeySetGroup** command displays the settings of a particular key set group.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set group. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

listKeySetGroups

The **listKeySetGroups** command lists the key set groups for a particular scope.

Target object

None.

Required parameters

None.

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-displayObjectNames
If you set the value of this parameter to true, the command returns a list of all of the key set group objects within a scope. If you set the value of this parameter to false, the command returns a list of strings that contain the key set group name and management scope. (Boolean, optional)

-all
Specify the value of this parameter as `true` to list all key set groups. This parameter overrides the scopeName parameter. The default value is `false`. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

modifyKeySetGroup

The **modifyKeySetGroup** command changes the settings of an existing key set group.

Target object

None.

Required parameters

-name
Specifies the name that uniquely identifies the key set group. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-autoGenerate
Set the value of this parameter to true if you want to automatically generate keys. If not, set the value to false. (Boolean, optional)

-wsScheduleName
Specifies the name of the scheduler to use to perform key generation. (String, optional)

-keySetObjectNames
A list of key set configuration names separated by colons (:). (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Creating a key set group configuration
A key set group manages one or more key sets. WebSphere Application Server uses key set groups to automatically generate cryptographic keys or multiple synchronized key sets.

Related reference

Key set groups settings
Use this page to create new key set groups.

DynamicSSLConfigSelections command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the DynamicSSLConfigSelections group can be used to create, delete, and query dynamic SSL configuration selection objects.

The DynamicSSLConfigSelections command group for the AdminTask object includes the following commands:

- "deleteDynamicSSLConfigSelection"
- "getDynamicSSLConfigSelection" on page 694
- "listDynamicSSLConfigSelections" on page 694

deleteDynamicSSLConfigSelection

The **deleteDynamicSSLConfigSelection** command deletes the dynamic SSL configuration selection from the configuration.

Target object

None.

Required parameters

-dynSSLConfigSelectionName
Specifies the name that uniquely identifies the dynamic SSL configuration selection. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteDynamicSSLConfigSelection {-dynSSLConfigSelectionName sampleConfigSelection}
```

- Using Jython:

```
AdminTask.deleteDynamicSSLConfigSelection(-dynSSLConfigSelectionName sampleConfigSelection)
```

-

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteDynamicSSLConfigSelection {-interactive}
```

- Using Jython:

```
AdminTask.deleteDynamicSSLConfigSelection('-interactive')
```

getDynamicSSLConfigSelection

The **getDynamicSSLConfigSelection** command obtains information about a particular dynamic SSL configuration selection.

Target object

None.

Required parameters

-dynSSLConfigSelectionName
Specifies the name that uniquely identifies the dynamic SSL configuration selection. (String, required)

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getDynamicSSLConfigSelection {-dynSSLConfigSelectionName sampleConfigSelection}
```

- Using Jython:

```
AdminTask.getDynamicSSLConfigSelection(-dynSSLConfigSelectionName sampleConfigSelection)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getDynamicSSLConfigSelection {-interactive}
```

- Using Jython:

```
AdminTask.getDynamicSSLConfigSelection('-interactive')
```

listDynamicSSLConfigSelections

The **listDynamicSSLConfigSelections** command lists the configuration objects name for a dynamic SSL configuration selection.

Target object

None.

Required parameters

None.

Optional parameters

-scopeName
Specifies the unique name that identifies the management scope. (String, optional)

-all
Specify the value of this parameter as `true` to list all dynamic SSL configuration selections. This parameter overrides the scopeName parameter. The default value is `false`. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listDynamicSSLConfigSelections
  ```
- Using Jython:
  ```
  AdminTask.listDynamicSSLConfigSelections()
  ```
- Using Jacl:
  ```
  $AdminTask listDynamic SSLConfigSelections
  ```
- Using Jython:
  ```
  AdminTask.listDynamic SSLConfigSelections()
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listDynamicSSLConfigSelections {-interactive}
  ```
- Using Jython:
  ```
  AdminTask.listDynamicSSLConfigSelections('-interactive')
  ```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Replacing an existing self-signed certificate
Occasionally, you need to replace an existing or expired self-signed certificate with a new certificate. Certificates are referenced in the runtime configuration by the Secure Sockets Layer (SSL) Configuration object and the Dynamic SSL Configuration Selection object. You can replace a certificate with a new certificate alias reference or with a new signer certificate.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

PersonalCertificateCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the PersonalCertificateCommands group can be used to create and manage personal or signer certificates.

The PersonalCertificateCommands command group for the AdminTask object includes the following commands:

createChainedCertificate

The createChainedCertificate command creates a new self-signed certificate and stores the certificate in a keystore.

Note: To use the IBMi5OSKeyStore key store, verify that the signer for each part of the chain exists in the keystore before creating the new certificate. You must import the signer into the IBMi5OSKeyStore keystore before creating the new certificate.

Target object

None.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
Specifies the name that uniquely identifies the certificate request in a keystore. (String, required)

-certificateSize
Specifies the size of the certificate. (Integer, required)

-certificateCommonName
Specifies the common name of the certificate. (String, required)

-certificateOrganization
Specifies the organization of the certificate. (String, optional)

Optional parameters

-rootCertificateAlias
Specifies a unique name to identify the root certificated to use for signing. The default root certificate alias is `root`. (String, optional)

-certificateVersion
Specifies the version of the certificate. (String, optional)

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-certificateOrganization
Specifies the organization of the certificate. (String, optional)

-certificateOrganizationalUnit
Specifies the organizational unit of the certificate. (String, optional)

-certificateLocality
Specifies the locality of the certificate. (String, optional)

-certificateState
Specifies the state of the certificate. (String, optional)

-certificateZip
Specifies the zip code of the certificate. (String, optional)

-certificateCountry
Specifies the country of the certificate. (String, optional)

-certificateValidDays
Specifies the amount of time in days for which the certificate is valid. (Integer, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createChainedCertificate('-keyStoreName myKeystore -certificateAlias
newCertificate -certificateSize 10 -certificateCommonName localhost
-certificateOrganization ibm')
```

- Using Jython list:

```
AdminTask.createChainedCertificate('-keyStoreName', 'myKeystore', '-certificateAlias',
'newCertificate', '-certificateSize', '10', '-certificateCommonName', 'localhost',
'-certificateOrganization', 'ibm')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createChainedCertificate('-interactive')
```

createSelfSignedCertificate

The createSelfSignedCertificate command creates a self-signed personal certificate in a keystore.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

-certificateVersion
The version of the certificate. (String, required)

-certificateSize
The size of the certificate. (Integer, required)

-certificateCommonName
The common name of the certificate. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

-certificateOrganization
The organization of the certificate. (String, optional)

-certificateOrganizationalUnit
The organizational unit of the certificate. (String, optional)

-certificateLocality
The locality of the certificate. (String, optional)

-certificateState
The state of the certificate. (String, optional)

-certificateZip
The zip code of the certificate. (String, optional)

-certificateCountry
The country of the certificate. (String, optional)

-certificateValidDays
The amount of time in days for which the certificate is valid. (Integer, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createSelfSignedCertificate {-keyStoreName testKeyStore -certificateAlias
default -certificateCommonName localhost -certificateOrganization ibm}
```

- Using Jython string:

```
AdminTask.createSelfSignedCertificate('[-keyStoreName testKeyStore -certificateAlias
default -certificateCommonName localhost -certificateOrganization ibm]')
```

- Using Jython list:

```
AdminTask.createSelfSignedCertificate(['-keyStoreName', 'testKeyStore', '-certificateAlias',
'default', '-certificateCommonName', 'localhost', '-certificateOrganization', 'ibm'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.createSelfSignedCertificate('-interactive')
```

deleteCertificate

The deleteCertificate command deletes a personal certificate from a keystore. The command saves a copy of the certificate in the delete keystore.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Interactive mode example usage:

- Using Jython:

```
AdminTask.deleteCertificate('-interactive')
```

exportCertificate

The exportCertificate command exports a personal certificate from one keystore to another.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-keyStorePassword
The password to the keystore. (String, required)

-keyFilePath
The full path to a keystore file that is located in a file system. The store from where a certificate will be imported or exported. (String, required)

-keyFilePassword
The password to the keystore file. (String, required)

-keyFileType
The type of the key file. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

-aliasInKeyStore
(String, optional)

Example output

The command does not return output.

Examples

Interactive mode example usage:

- Using Jython:

```
AdminTask.exportCertificate('-interactive')
```

exportCertToManagedKS

The exportCertToManagedKS command exports a personal certificate to a managed keystore in the configuration.

Target object

None.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore configuration object. (String, required)

-keyStorePassword
The password to the keystore. (String, required)

-toKeyStoreName
Specifies the unique name of the keystore to export the certificate to. (String, required)

-certificateAlias
Specifies the alias of the certificate of interest. (String, required)

Optional parameters

-keyStoreScope
Specifies the keystore of the certificate of interest. (String, optional)

-toKeyStoreScope
Specifies the scope of the keystore to export to. (String, optional)

-aliasInKeyStore
Specifies the alias that identifies the certificate in the keystore. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportCertificateToManagedKS('-keyStoreName myKS -keyStorePassword myKSpw
-toKeyStoreName myKS2 -certificateAlias testingKeyStore')
```

- Using Jython list:

```
AdminTask.exportCertificateToManagedKS(['-keyStoreName', 'myKS', '-keyStorePassword',
'myKSpw', '-toKeyStoreName', 'myKS2', '-certificateAlias', 'testingKeyStore'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportCertificateToManagedKS('-interactive')
```

extractCertificate

The extractCertificate command extracts the signer part of a personal certificate to a certificate file. The certificate in the file can later be added to a keystore to establish trust.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

-certificateFilePath
The full path of the request file that contains the certificate. (String, required)

-base64Encoded
Set the value of this parameter to `true` if the certificate is a Base64 encoded ASCII file type. Set the value of this parameter to `false` if the certificate is binary. (Boolean, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask extractCertificate {-keyStoreName testKeyStore -certificateFilePath
/temp/CertFile.arm -certificateAlias testCertificate}
```

- Using Jython string:

```
AdminTask.extractCertificate('[-keyStoreName testKeyStore -certificateFilePath
/temp/CertFile.arm -certificateAlias testCertificate]')
```

- Using Jython list:

```
AdminTask.extractCertificate(['-keyStoreName', 'testKeyStore', '-certificateFilePath',
'/temp/CertFile.arm', '-certificateAlias', 'testCertificate'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.extractCertificate('-interactive')
```

getCertificate

The getCertificate command obtains information about a particular personal certificate in a keystore. If the certificate of interest was created with the requestCACertificate command, the certificate can be in the `COMPLETE` or `REVOKED` state. Certificate requests can be in the `PENDING` state. Use the getCertificateRequest command to determine if a certificate request is in the `PENDING` state.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command returns information about the certificate request.

Examples

Interactive mode example usage:

- Using Jython:

```
AdminTask.getCertificate('-interactive')
```

getCertificateChain

The getCertificateChain command queries your configuration for information about each personal certificate in a certificate chain.

Target object

None.

Required parameters and return values

-keyStoreName
Specifies the name of the keystore object that stores the CA certificate. Use the listKeyStores command to display a list of available keystores. (String, required)

-certificateAlias
Specifies the unique alias of the certificate. (String, required)

Optional parameters

-keyStoreScope
Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope. (String, optional)

Example output

The command returns an array of attribute lists that contain configuration information for each certificate in a chain.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getCertificateChain {-certificateAlias newCertificate
-keyStoreName CellDefaultKeyStore}
```

- Using Jython string:

```
AdminTask.getCertificateChain('-certificateAlias newCertificate
-keyStoreName CellDefaultKeyStore')
```

- Using Jython list:

```
AdminTask.getCertificateChain(['-certificateAlias', 'newCertificate',
'-keyStoreName', 'CellDefaultKeyStore'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.getCertificateChain('-interactive')
```

importCertificate

The importCertificate command imports a personal certificate from a keystore.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-keyFilePath
The full path to a keystore file that is located in a file system. The store from where a certificate will be imported or exported. (String, required)

-keyFilePassword
The password to the keystore file. (String, required)

-keyFileType
The type of the key file. (String, required)

-certificateAliasFromKeyFile
The certificate alias in the key file from which the certificate is being imported. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Interactive mode example usage:

- Using Jython:

```
AdminTask.importCertificate('-interactive')
```

importCertFromManagedKS

The importCertFromManagedKS command imports a personal certificate from a managed keystore in the configuration.

Target object

None.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore configuration object. (String, required)

-fromKeyStoreName
Specifies the name that uniquely identifies the keystore from which the system imports the certificate. (String, required)

-fromKeyStorePassword
Specifies the password for the keystore from which the system imports the certificate. (String, required)

-certificateAliasFromKeyStore
Specifies the alias of the certificate in the keystore. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope of the keystore to import the certificate to. (String, optional)

-fromKeyStoreScope
Specifies the scope of the keystore to import the certificate from. (String, optional)

-certificateAlias
Specifies the alias of the certificate for the destination keystore. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.importCertFromManagedKS('-keyStoreName myKeystore -fromKeyStoreName
oldKeystore -fromKeyStorePassword my122password -certificateAliasFromKeyStore
myCertificate')
```

- Using Jython list:

```
AdminTask.importCertFromManagedKS('-keyStoreName', 'myKeystore', '-fromKeyStoreName',
'oldKeystore', '-fromKeyStorePassword', 'my122password', '-certificateAliasFromKeyStore',
'myCertificate')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importCertFromManagedKS('-interactive')
```

listPersonalCertificates

The listPersonalCertificates command lists the personal certificates in a particular keystore.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command returns a list of attributes for each personal certificate in a keystore.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.listPersonalCertificates('-keyStoreName myKS')
```

- Using Jython list:

```
AdminTask.listPersonalCertificates(['-keyStoreName', 'myKS'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.listPersonalCertificates('-interactive')
```

queryCACertificate

The queryCACertificate command queries your configuration to determine if the CA has completed the certificate. If the CA returns a personal certificate, then the system marks the certificate as `COMPLETE`. Otherwise, it remains marked as `PENDING`.

Target object

None.

Required parameters and return values

-keyStoreName
Specifies the name of the keystore object that stores the CA certificate. Use the listKeyStores command to display a list of available keystores. (String, required)

-certificateAlias
Specifies the unique alias of the certificate. (String, required)

Optional parameters

-keyStoreScope
Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope. (String, optional)

Example output

The command returns one of two values: `Certificate COMPLETE` or `certificate PENDING`. If the command returns the `Certificate COMPLETE` message, the certificate authority returned the requested certificate and the default personal certificate is replaced. If the command returns the `certificate PENDING` message, the certificate authority did not yet return a certificate.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask queryCACertificate {-certificateAlias newCertificate
-keyStoreName CellDefaultKeyStore}
```

- Using Jython string:

```
AdminTask.queryCACertificate('-certificateAlias newCertificate
-keyStoreName CellDefaultKeyStore')
```

- Using Jython list:

```
AdminTask.queryCACertificate(['-certificateAlias', 'newCertificate',
'-keyStoreName', 'CellDefaultKeyStore'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.queryCACertificate('-interactive')
```

receiveCertificate

The receiveCertificate command receives a signer certificate from a file to a personal certificate.

Target object

None.

Required parameters

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

-certificateFilePath
The full path of the file that contains the certificate. (String, required)

-base64Encoded
Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (Boolean, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask receiveCertificate {-keyStoreName testKeyStore
-certificateFilePath /temp/CertFile.arm}
```

- Using Jython string:

```
AdminTask.receiveCertificate('[-keyStoreName testKeyStore
-certificateFilePath /temp/CertFile.arm]')
```

- Using Jython list:

```
AdminTask.receiveCertificate(['-keyStoreName', 'testKeyStore',
'-certificateFilePath', '/temp/CertFile.arm'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.receiveCertificate('-interactive')
```

renewCertificate

The renewCertificate command renews a certificate with a new generated certificate.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name that identifies the keystore. (String, required)

-certificateAlias
Specifies the unique name that identifies the certificate. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope of the keystore. (String, optional)

-deleteOldSigners
Specifies whether to delete the old signers that are associated with the old certificate. Specify `false` to retain the old signers. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.renewCertificate('-keyStoreName myKS -certificateAlias
testCertificate')
```

- Using Jython list:

```
AdminTask.renewCertificate(['-keyStoreName', 'myKS', '-certificateAlias',
'testCertificate'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.renewCertificate('-interactive')
```

replaceCertificate

The replaceCertificate command replaces a personal certificate with another personal certificate. The command finds each reference to the old certificate alias in the configuration and replaces the alias with the new one. The command also replaces each signer certificate from the old personal certificate with the signer from the new personal certificate.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
The name that uniquely identifies the certificate request in a keystore. (String, required)

-replacementCertificateAlias
The alias of the certificate that is used to replace a different certificate. (String, required)

Optional parameters

-keyStoreScope
The scope name of the keystore. (String, optional)

-deleteOldCert
Set the value of this parameter to `true` if you want to delete the old signer certificates during certificate replacement. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

-deleteOldSigners
Set the value of this parameter to `true` if you want to delete the old certificates during certificate replacement. Otherwise, set the value of this parameter to `false`. (Boolean, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask replaceCertificate {-keyStoreName testKeyStore -certificateAlias
default -replacementCertificateAlias replaceCert -deleteOldCert true
-deleteOldSigners true}
```

- Using Jython string:

```
AdminTask.replaceCertificate('[-keyStoreName testKeyStore -certificateAlias
default -replacementCertificateAlias replaceCert -deleteOldCert true
-deleteOldSigners true]')
```

- Using Jython list:

```
AdminTask.replaceCertificate(['-keyStoreName', 'testKeyStore', '-certificateAlias',
'default', '-replacementCertificateAlias', 'replaceCert', '-deleteOldCert',
'true', '-deleteOldSigners', 'true'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.replaceCertificate('-interactive')
```

requestCACertificate

The requestCACertificate command creates a certificate request and sends the request to a certificate authority (CA). If the certificate authority returns a personal certificate, then the returned certificate replaces the certificate request in the keystore. The command also works with a preexisting certificate request that was created with the createCertificateRequest command. When the CA returns a personal certificate, the system marks the certificate as `COMPLETE` and the command returns a message stating that the certificate is complete. If the CA does not return a personal certificate, then the system marks the certificate request as `PENDING` and the command returns a message stating that the certificate is `PENDING`.

Note: To use the IBMi5OSKeyStore key store, verify that the signer for each part of the chain exists in the keystore before creating the new certificate. You must import the signer into the IBMi5OSKeyStore keystore before creating the new certificate.

Target object

None.

Required parameters and return values

-certificateAlias
Specifies the alias of the certificate. You can specify a predefined certificate request. (String, required)

-keyStoreName
Specifies the name of the keystore object that stores the CA certificate. Use the listKeyStores command to display a list of available keystores. (String, required)

-caClientName
Specifies the name of the CA client that was used to create the CA certificate. (String, required)

-revocationPassword
Specifies the password to use to revoke the certificate at a later date. (String, required)

Optional parameters

-keyStoreScope
Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope. (String, optional)

-caClientScope
Specifies the management scope of the CA client. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope. (String, optional)

-certificateCommonName
Specifies the common name (CN) part of the full distinguished name (DN) of the certificate. This common name can represent a person, company, or machine. For Web sites, the common name is frequently the DNS host name where the server resides. (String, optional)

-certificateOrganization
Specifies the organization part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateOrganizationalUnity
Specifies the organization unit part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateLocality
Specifies the locality part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateState
Specifies the state part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateZip
Specifies the zip code part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateCountry
Specifies the country part of the full distinguished name (DN) of the certificate. (String, optional)

-certificateSize
Specifies the size of the certificate key. The valid values are 512, 1024, and 2048. The default value is 1024. (String, optional)

Example output

The command returns one of two values: `Certificate COMPLETE` or `certificate PENDING`.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask requestCACertificate {-certificateAlias newCertificate -keyStoreName
CellDefaultKeyStore -CAClientName myCAClient -revocationPassword revokeCApw}
```

- Using Jython string:

```
AdminTask.requestCACertificate('-certificateAlias newCertificate -keyStoreName
CellDefaultKeyStore -CAClientName myCAClient -revocationPassword revokeCApw')
```

- Using Jython list:

```
AdminTask.requestCACertificate(['-certificateAlias','newCertificate','-keyStoreName',
'CellDefaultKeyStore','-CAClientName','myCAClient','-revocationPassword',
'revokeCApw'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.requestCACertificate('-interactive')
```

revokeCACertificate

The revokeCACertificate command sends a request to the CA to revoke the CA personal certificate of interest.

Target object

None.

Required parameters and return values

-certificateAlias
Specifies the unique name that identifies the CA personal certificate object and the alias name of the certificate in the keystore. (String, required)

-keyStoreName
Specifies the name of the keystore where the CA personal certificate is stored. (String, required)

-revocationPassword
Specifies the password needed to revoke the certificate. This is the same password that was provided when the certificate was created. (String, required)

Optional parameters

-keyStoreScope
Specifies the management scope of the keystore. For a deployment manager profile, the default value is the cell scope. For an application server profile, the default value is the node scope. (String, optional)

-revocationReason
Specifies the reason for revoking the certificate of interest. The default value for this parameter is `unspecified`. (String, optional)

Example output

The command does not return output. Use the getCertificate command to view the current status of the certificate, as the following example displays:

```
AdminTask.getCertificate('-certificateAlias myCertificate -keyStoreName CellDefaultKeyStore')
```

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask revokeCACertificate {-keyStoreName CellDefaultKeyStore -certificateAlias
myCertificate -revocationPassword pw4revoke}
```

- Using Jython string:

```
AdminTask.revokeCACertificate('[-keyStoreName CellDefaultKeyStore -certificateAlias
myCertificate -revocationPassword pw4revoke]')
```

- Using Jython list:

```
AdminTask.revokeCACertificate(['-keyStoreName', 'CellDefaultKeyStore', '-certificateAlias',
'myCertificate', '-revocationPassword', 'pw4revoke'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.revokeCACertificate('-interactive')
```

WSCertExpMonitorCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the WSCertExpMonitorCommands group can be used to start or update the certificate expiration monitor.

The WSCertExpMonitorCommands command group for the AdminTask object includes the following commands:

- “createWSCertExpMonitor”
- “deleteWSCertExpMonitor” on page 713
- “getWSCertExpMonitor” on page 713
- “listWSCertExpMonitor” on page 714
- “modifyWSCertExpMonitor” on page 714
- “startCertificateExpMonitor” on page 715

createWSCertExpMonitor

The **createWSCertExpMonitor** command creates the certificate expiration monitor settings in the configuration.

Target object

None.

Required parameters and return values

-name
The name that uniquely identifies the certificate expiration monitor. (String, required)

-autoReplace
Set the value of this parameter to true if you want to replace a certificate within a certificate expiration date. If not, set the value of this parameter to false. (Boolean, required)

-deleteOld
Set the value of this parameter to true if you want to delete an old certificate during certificate expiration monitoring. If not, set the value of this parameter to false. (Boolean, required)

-daysBeforeNotification
The number of days before a certificate expires that you want to be notified of the expiration. (Integer, required)

-wsScheduleName
The name of the scheduler to use for certificate expiration. (String, required)

-wsNotificationName
The name of the notifier to use for certificate expiration. (String, required)

-isEnabled
Set the value of this parameter to true if the certificate expiration monitor is enabled. If not, set the value of this parameter to false. (Boolean, optional)

- Returns: The configuration object name of the certificate expiration monitor object that you created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWSCertExpMonitor {-name testCertMon –autoReplace true –deleteOld true
–daysBeforeNotification 30 –wsScheduleName testSchedule –wsNotificationName testNotifier
–isEnabled false}
```

- Using Jython string:

```
AdminTask.createWSCertExpMonitor ('[-name testCertMon –autoReplace true –deleteOld true
–daysBeforeNotification 30 –wsScheduleName testSchedule –wsNotificationName testNotifier
–isEnabled false]')
```

- Using Jython list:

```
AdminTask.createWSCertExpMonitor (['-name', 'testCertMon', '–autoReplace', 'true', '–deleteOld',
'true', '–daysBeforeNotification', '30', '–wsScheduleName', 'testSchedule', '–wsNotificationName',
'testNotifier', '–isEnabled', 'false'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWSCertExpMonitor {-interactive}
```

- Using Jython string:

```
AdminTask.createWSCertExpMonitor ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createWSCertExpMonitor (['-interactive'])
```

deleteWSCertExpMonitor

The **deleteWSCertExpMonitor** command deletes the settings of a scheduler from the configuration.

Target object

None.

Required parameters and return values

-name
The name that uniquely identifies the certificate expiration monitor. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteWSCertExpMonitor {-name testCertMon}
  ```
- Using Jython string:
  ```
  AdminTask.deleteWSCertExpMonitor ('[-name testCertMon]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteWSCertExpMonitor (['-name', 'testCertMon'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteWSCertExpMonitor {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.deleteWSCertExpMonitor ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteWSCertExpMonitor (['-interactive'])
  ```

getWSCertExpMonitor

The **getWSCertExpMonitor** command displays the settings of a particular scheduler.

Target object

None.

Required parameters and return values

-name
The name that uniquely identifies the certificate expiration monitor. (String, required)

- Returns: The scheduler in the configuration.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getWSCertExpMonitor {-name testCertMon}
  ```
- Using Jython string:
  ```
  AdminTask getWSCertExpMonitor ('[-name testCertMon]')
  ```
- Using Jython list:

```
AdminTask getWSCertExpMonitor (['-name', 'testCertMon'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask getWSCertExpMonitor {-interactive}
```
- Using Jython string:
```
AdminTask.getWSCertExpMonitor ('[-interactive]')
```
- Using Jython list:
```
AdminTask.getWSCertExpMonitor (['-interactive'])
```

listWSCertExpMonitor

The **listWSCertExpMonitor** command lists the scheduler in the configuration.

Target object

None.

Required parameters and return values

-displayObjectNames
If you set the value of this parameter to true, the command returns the certificate expiration monitor configuration object. If you set the value of this parameter to false, the command returns the name of the certificate expiration monitor. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask listWSCertExpMonitor {-displayObjectName false}
```
- Using Jython string:
```
AdminTask.listWSCertExpMonitor ('[-displayObjectName false]')
```
- Using Jython list:
```
AdminTask.listWSCertExpMonitor (['-displayObjectName', 'false'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask listWSCertExpMonitor {-interactive}
```
- Using Jython string:
```
AdminTask.listWSCertExpMonitor ('[-interactive]')
```
- Using Jython list:
```
AdminTask.listWSCertExpMonitor (['-interactive'])
```

modifyWSCertExpMonitor

The **modifyWSCertExpMonitor** command changes the setting of an existing scheduler.

Target object

None.

Required parameters and return values

-name
The name that uniquely identifies the certificate expiration monitor. (String, required)

-autoReplace
Set the value of this parameter to true if you want to replace a certificate within a certificate expiration date. If not, set the value of this parameter to false. (Boolean, required)

-deleteOld
Set the value of this parameter to true if you want to delete an old certificate during certificate expiration monitoring. If not, set the value of this parameter to false. (Boolean, required)

-daysBeforeNotification
The number of days before a certificate expires that you want to be notified of the expiration. (Integer, required)

-wsScheduleName
The name of the scheduler to use for certificate expiration. (String, required)

-wsNotificationName
The name of the notifier to use for certificate expiration. (String, required)

-isEnabled
Set the value of this parameter to true if the certificate expiration monitor is enabled. If not, set the value of this parameter to false. (Boolean, optional)

- Returns: None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyWSCertExpMonitor {-name testCertMon –autoReplace false –deleteOld false
–daysBeforeNotification 20 –isEnabled true}
```

- Using Jython string:

```
AdminTask.modifyWSCertExpMonitor ('[-name testCertMon –autoReplace false –deleteOld false
–daysBeforeNotification 20 –isEnabled true]')
```

- Using Jython list:

```
AdminTask.modifyWSCertExpMonitor (['-name', 'testCertMon', '–autoReplace', 'false', '–deleteOld',
'false', '–daysBeforeNotification', '20', '–isEnabled', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyWSCertExpMonitor {-interactive}
```

- Using Jython string:

```
AdminTask.modifyWSCertExpMonitor ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifyWSCertExpMonitor (['-interactive'])
```

startCertificateExpMonitor

The **startCertificateExpMonitor** command performs certificate monitoring. This command visits all key stores and checks to see if they are within certificate expiration range.

Target object

None.

Required parameters and return values

- Parameters: None
- Returns: None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask startCertificateExpMonitor
```

- Using Jython:

```
AdminTask.startCertificateExpMonitor()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask startCertificateExpMonitor {-interactive}
```

- Using Jython string:

```
AdminTask.startCertificateExpMonitor ('[-interactive]')
```

- Using Jython list:

```
AdminTask.startCertificateExpMonitor (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

SignerCertificateCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the SignerCertificateCommands group can be used to create and modify signer certificates in relation to the key store file and to query for signer information on ports of remote hosts.

The SignerCertificateCommands command group for the AdminTask object includes the following commands:

- "addSignerCertificate"
- "deleteSignerCertificate" on page 717
- "extractSignerCertificate" on page 718
- "getSignerCertificate" on page 719
- "listSignerCertificates" on page 719
- "retrieveSignerFromPort" on page 720
- "retrieveSignerInfoFromPort" on page 721

addSignerCertificate

The **addSignerCertificate** command add a signer certificate from a certificate file to a keystore.

Target object

None.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
Specifies the name that uniquely identifies the certificate request in a keystore. (String, required)

-certificateFilePath
Specifies the full path of the request file that contains the certificate. (String, required)

-base64Encoded
Specifies that the certificate is a Base64 encoded ASCII data file type if the value is set to `true`. Set the value of this parameter to `false` if the certificate is a binary DER data file type. (Boolean, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addSignerCertificate {-keyStoreName testKeyStore -certificateAlias
default -certificateFilePath <file path> -base64Encoded true}
```

- Using Jython string:

```
AdminTask.addSignerCertificate('[-keyStoreName testKeyStore -certificateAlias
default -certificateFilePath <file path> -base64Encoded true]')
```

- Using Jython list:

```
AdminTask.addSignerCertificate(['-keyStoreName', 'testKeyStore', '-certificateAlias',
'default', '-certificateFilePath', '<file path>', '-base64Encoded', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addSignerCertificate {-interactive}
```

- Using Jython string:

```
AdminTask.addSignerCertificate ('[-interactive]')
```

deleteSignerCertificate

The **deleteSignerCertificate** command delete a signer certificate from a certificate file from a keystore.

Target object

None.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore configuration object. (String, required)

-certificateAlias
Specifies the name that uniquely identifies the certificate request in a keystore. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

Example output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteSignerCertificate {-keyStoreName testKeyStore -certificateAlias
default}
```

- Using Jython string:

```
AdminTask.deleteSignerCertificate('[-keyStoreName testKeyStore -certificateAlias
default]')
```

- Using Jython list:

```
AdminTask.deleteSignerCertificate(['-keyStoreName', 'testKeyStore', '-certificateAlias',
'default'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteSignerCertificate {-interactive}
```

- Using Jython string:

```
AdminTask.deleteSignerCertificate ('[-interactive]')
```

extractSignerCertificate

The **extractSignerCertificate** command extracts a signer certificate from a key store to a file.

Target object

None

Required parameters and return values

-keyStoreName
The name of the key store where the signer certificate is located. (String, required)

-keyStoreScope
The management scope of the key store. (String, optional)

-certificateAlias
The alias name of the signer certificate in the key store. (String, required)

-certificateFilePath
The full path name of the file that contains the signer certificate. (String, required)

-base64Encoded
Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (String, required)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask extractSigner Certificate {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.extractSignerC ertificate ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.extractSignerCe rtificate (['-interactive'])
  ```

getSignerCertificate

The **getSignerCertificate** command obtains information about a signer certificate from a key store.

Target object

None

Required parameters and return values

-keyStoreName
The name of the key store where the signer certificate is located. (String, required)

-keyStoreScope
The management scope of the key store. (String, optional)

-certificateAlias
The alias name of the signer certificate in the key store. (String, required)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getSignerCert ificate {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getSignerCert ificate ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getSignerCert ificate (['-interactive'])
  ```

listSignerCertificates

The **listSignerCertificates** command lists all signer certificates in a particular key store.

Target object

None

Required parameters and return values

-keyStoreName
The name of the key store where the signer certificate is located. (String, required)

-keyStoreScope
The management scope of the key store. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listSignerCer tificates {-interactive}
```

- Using Jython string:

```
AdminTask.listSignerCer tificates ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listSignerCer tificates (['-interactive'])
```

retrieveSignerFromPort

The **retrieveSignerFromPort** command retrieves a signer from a remote host and stores the signer in a key store.

Target object

None

Required parameters and return values

-host
The host name of the system from where the signer certificate will be retrieved. (String, required)

-port
The port of the remote system from where the signer certificate will be retrieved. (Integer, required)

-keyStoreName
The name of the key store where the signer certificate is located. (String, required)

-keyStoreScope
The management scope of the key store. (String, required)

-sslConfigName
The name of the SSL configuration object. (String, optional)

-sslConfigScopeName
The management scope where the SSL configuration object is located. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask retrieveSigner FromPort {-host serverHost -port 443 -keyStoreName testKeyStore
-certificate Alias serverHostSigner}
```

- Using Jython string:

```
AdminTask.retrieveSigner FromPort ('[-host server Host -port 443 -keyStore Name testKeyStore
-certi ficateAlias serverHost Signer]')
```

- Using Jython list:

```
AdminTask.retrieveSigner FromPort (['-host', 'serverHost', '-port', '443', '-keyStoreName',
'testKeyStore', '-certi ficateAlias', 'serverHost Signer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask retrieveSigner FromPort {-interactive}
```

- Using Jython string:

```
AdminTask.retrieveSigner FromPort ('[-interactive]')
```

- Using Jython list:

```
AdminTask.retrieveSigner FromPort (['-interactive'])
```

retrieveSignerInfoFromPort

The **retrieveSigner InfoFromPort** command retrieves signer information from a port on a remote host.

Target object

None

Required parameters and return values

-host
The host name of the system from where the signer certificate will be retrieved. (String, required)

-port
The port of the remote system from where the signer certificate will be retrieved. (Integer, required)

-sslConfigName
The name of the SSL configuration object. (String, optional)

-sslConfigScopeName
The management scope where the SSL configuration object is located. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask retrieveSigner InfoFromPort {-interactive}
```

- Using Jython string:

```
AdminTask.retrieveSigner InfoFromPort ('[-interactive]')
```

- Using Jython list:

```
AdminTask.retrieveSigner InfoFromPort (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

CertificateRequestCommands command group of the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the CertificateRequestCommands group can be used to create and manage certificate requests.

The CertificateRequestCommands command group for the AdminTask object includes the following commands:

createCertificateRequest

The **createCertificateRequest** command creates a certificate request that is associated with a particular key store.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the key store configuration object. (String, required)

-keyStoreScope
The scope name of the key store. (String, optional)

-certificateAlias
The name that uniquely identifies the certificate request in a key store. (String, required)

-certificateVersion
The certificate version. (String, required)

-certificateSize
(Integer, required)

-certificateCommonName
(String, required)

-certificateOrganization
(String, optional)

-certificateOrganizationalUnit
(String, optional)

-certificateLocality
(String, optional)

-certificateState
The state code for the certificate. (String, optional)

-certificateZip
The zip code for the certificate. (String, optional)

-certificateCountry
The country for the certificate. (String, optional)

-certificateValidDays
The amount of time in days for which the certificate is valid. (Integer, optional)

-certificateRequestFilePath
The file location of the certificate request that can be sent to a certificate authority. (String, required)

- Returns: The configuration object name of the key store object that you created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createCertificateRequest {-keyStoreName testKeyStore
-certificateAlias certReq -certificateSize 1024 -certificate
CommonName localhost -certificate Organization testing -certificate
RequestFilePath c:\temp\testCertReq.arm}
```

- Using Jython string:

```
AdminTask.createCertificateRequest ('[-keyStoreName testKeyStore
-certificateAlias certReq -certificateSize 1024 -certificate
CommonName localhost -certificate Organization testing -certificate
RequestFilePath c:\temp\testCertReq.arm]')
```

- Using Jython list:

```
AdminTask.createCertificateRequest (['-keyStoreName', 'testKeyStore',
'-certificateAlias', 'certReq', '-certificateSize', '1024',
'-certificateCommonName', 'localhost','-certificateOrganization',
'testing', '-certificateRequestFilePath','c:\temp\testCertReq.arm'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCertificateRequest {-interactive}
```

- Using Jython string:

```
AdminTask.createCertificateRequest ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createCertificateRequest (['-interactive'])
```

deleteCertificateRequest

The **deleteCertificateRequest** command deletes a certificate request from a key store.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the key store configuration object. (String, required)

-keyStoreScope
The scope name of the key store. (String, optional)

-certificateAlias
The name that uniquely identifies the certificate request in a key store. (String, required)

- Returns: None.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCertificateRequest {-interactive}
```

- Using Jython string:

```
AdminTask.deleteCertificateRequest ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteCertificateRequest (['-interactive'])
```

extractCertificateRequest

The **extractCertificateRequest**command extracts a certificate request to a file.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the key store configuration object. (String, required)

-keyStoreScope
The scope name of the key store. (String, optional)

-certificateAlias
The name that uniquely identifies the certificate request in a key store. (String, required)

-certificateRequestFilePath
The file location of the certificate request that can be sent to a certificate authority. (String, required)

- Returns: A certificate request file is created that contains the extracted certificate.

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask extractCertificateRequest {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.extractCertificateRequest ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.extractCertificateRequest (['-interactive'])
  ```

getCertificateRequest

The **getCertificateRequest** command obtains information about a particular certificate request in a key store.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the key store configuration object. (String, required)

-keyStoreScope
The scope name of the key store. (String, optional)

-certificateAlias
The name that uniquely identifies the certificate request in a key store. (String, required)

- Returns: Information about the certificate request.

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getCertificateRequest {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getCertificateRequest ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getCertificateRequest (['-interactive'])
  ```

listCertificateRequest

The **listCertificateRequest** command lists all the certificate requests associated with a particular key store.

Target object

None.

Required parameters and return values

-keyStoreName
The name that uniquely identifies the key store configuration object. (String, required)

-keyStoreScope
The scope name of the key store. (String, optional)

- Returns: An attribute list for each certificate request in a key store.

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listCertificateRequest {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listCertificateRequest ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listCertificateRequest (['-interactive'])
  ```

Enabling authentication in the file transfer service using scripting

You can enable authentication in the file transfer service using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

In WebSphere Application Server Network Deployment, V5.0.1 or later, the file transfer service is enhanced to provide role-based authentication. Two versions of the file transfer Web application are provided. By default, the version that does not authenticate its caller is installed. This default supports compatibility between the WebSphere Application Server Network Deployment, V5.0 and V5.0.1 or later.

Turning the file transfer authentication on is recommended to prevent unauthorized use of the file transfer application; however, if you have any V5.0 clients in your Network Deployment environment, they cannot communicate with the secured file transfer application if global security is turned on.

In WebSphere Application Server V6.x, mixed cells are supported and file transfer has become a system application. If all of the nodes in the cell are of V5.0.1 or later, you can activate authentication in the file transfer service by redeploying the file transfer application at the deployment manager. The compatible version is shipped in the *app_server_root*/`systemApps/filetransfer.ear` directory. The secured version is provided in the *app_server_root*/`systemApps/filetransferSecured.ear` directory.

- A wsadmin Jacl script is provided to help you redeploy the file transfer. The script is called redeployFileTransfer.jacl and is located in the *app_server_root*/`bin` directory.

After the deployment manager and all the nodes are upgraded to WebSphere Application Server Network Deployment, version 5.0.1 or later, you can deploy the secured file transfer service by running the script.

The syntax for running the script from the bin directory is the following:

–

```
wsadmin -conntype NONE -lang jacl -profile redeployFileTransfer.jacl -c "fileTransferAuthenticationXxx
cellName nodeName serverName"
```

where Xxx is **On** or **Off**.

Use wsadmin.

– For example, when running the script to enable use of the filetransferSecured.ear file, the syntax is similar to the following example:

```
wsadmin -conntype NONE -lang jacl -profile redeployFileTransfer.jacl -c
"fileTransferAuthenticationOn managedCell managedCellManager dmgr"
```

or

```
wsadmin -conntype NONE -lang jacl -profile redeployFileTransfer.jacl -c
"fileTransferAuthenticationOn baseCell base server1"
```

- If you want to go return to running the file transfer service without authentication, you can run the script as shown in the following example:

```
wsadmin -conntype NONE -lang jacl -profile redeployFileTransfer.jacl -c
"fileTransferAuthenticationOff baseNodeCell baseNode server1"
```

or

```
wsadmin -conntype NONE -lang jacl -profile redeployFileTransfer.jacl -c
"fileTransferAuthenticationOff managedCell managedCellManager dmgr"
```

What to do next

You must restart the server for the change to take affect.

Propagating security policy of installed applications to a JACC provider using wsadmin scripting

It is possible that you have applications installed prior to enabling the Java Authorization Contract for Containers (JACC)-based authorization. You can start with default authorization and then move to an external provider-based authorization using JACC later.

Before you begin

Note: Use the wsadmin tool to propagate information to the JACC provider independent of the application installation process, avoiding the need to reinstall applications. Also, during application installation or modification you might have had problems propagating the security policy information to the JACC provider. For example, network problems might occur, the JACC provider might not be available, and so on. For these cases, the security policy of the previously installed applications does not exist in the JACC provider to make the access decisions. One choice is to reinstall the applications involved. However, you can avoid reinstalling by using the wsadmin scripting tool. Use this tool to propagate information to the JACC provider independent of the application installation process. The tool eliminates the need for reinstalling the applications.

The tool uses the SecurityAdmin MBean to propagate the policy information in the deployment descriptor of any installed application to the JACC provider. You can invoke this tool using wsadmin at the base application server for base and deployment manager level for Network Deployment. Note that the SecurityAdmin MBean is available only when the server is running.

Use `propagatePolicyToJACCProvider{-appNames appNames}` to propagate the policy information in the deployment descriptor or annotations of the enterprise archive (EAR) files to the JACC provider. If the RoleConfigurationFactory and the RoleConfiguration interfaces are implemented by the JACC provider, the authorization table information in the binding file of the EAR files is also propagated to the provider. See the *Securing applications and their environment* PDF for more information about these interfaces.

The `appNames String` contains the list of application names, delimited by a colon (:), whose policy information must be stored in the provider. If appNames is not present, the policy information of all the deployed applications is propagated to the provider.

Also, be aware of the following items:

- Before migrating applications to the Tivoli Access Manager JACC provider, create or import the users and groups that are in the applications to Tivoli Access Manager.
- Depending on the application or the number of applications that are propagated, you might have to increase the request time-out period either in the `soap.client.props` file in the directory *profile_root*`/properties` (if using SOAP) or in the `sas.client.props` file (if using RMI) for the command to complete. You can set the request time-out value to 0 to avoid the timeout problem, and change it back to the original value after the command is run.

1. Configure your JACC provider in WebSphere Application Server.

   See the *Securing applications and their environment* PDF for more information.
2. Restart the server.
3. Enter the following commands:

   ```
   wsadmin>$AdminTask propagatePolicyToJACCProvider {-appNames appNames}
   ```

JACCUtilityCommands command group for the AdminTask object

Use this topic as a reference for the commands for the JACCUtilityCommands group for the AdminTask object. Use these commands to determine whether Java Authorization Contract for Containers (JACC) is enabled and whether the runtime uses a single security domain. You can also use these commands to propagate the security policies for application to the JACC provider.

The following commands are available for the JACCUtilityCommands group of the AdminTask object.

- "isJACCEnabled"
- "isSingleSecurityDomain" on page 728
- "propagatePolicyToJACCProvider " on page 728

isJACCEnabled

The isJACCEnabled command displays whether JACC is enabled or disabled in the global security domain when the server was started. The command does not indicate dynamic changes. Instead, it displays the JACC status at server startup.

Target object

None.

Required parameters

None.

Return value

The command returns `true` if JACC is enabled. The command returns `false` if JACC is disabled.

Batch mode example usage

Using Jython string:

```
AdminTask.isJACCEnabled()
```

Interactive mode example usage

Using Jython:

```
AdminTask.isJACCEnabled('-interactive')
```

isSingleSecurityDomain

The isSingleSecurityDomain command displays whether the environment is configured to use a single security domain when the server was started. The command does not indicate dynamic changes. Instead, it displays the security domain status at server startup.

Target object

None.

Required parameters

None.

Return value

The command returns `true` if the environment uses a single security domain. The command returns the `false` string if the environment uses multiple security domains.

Batch mode example usage

Using Jython:

```
AdminTask.isSingleSecurityDomain()
```

Interactive mode example usage

Using Jython:

```
AdminTask.isSingleSecurityDomain('-interactive')
```

propagatePolicyToJACCProvider

The propagatePolicyToJACCProvider command propagates the security policies of the applications of interest to the JACC provider. This command is supported in a single security domain environment only.

Target object

None.

Required parameters

None.

Optional parameters

-appNames
Specifies a list of application names delimited with a colon character (**:**). (String, optional)

The command uses all applications if you do not specify a value for this parameter, as the following syntax demonstrates: `AdminTask.propagatePolicyToJACCProvider()`

Return value

The command does not return output.

Batch mode example usage

Using Jython string:

```
AdminTask.propagatePolicyToJACCProvider ('–appNames "app1:app2:app3"')
```

Using Jython list:

```
AdminTask.propagatePolicyToJACCProvider ('–appNames', '"app1:app2:app3"')
```

Interactive mode example usage

Using Jython:

```
AdminTask.propagatePolicyToJACCProvider ('-interactive')
```

Configuring custom adapters for federated repositories using wsadmin

You can use the Jython or Jacl scripting language with the wsadmin tool to define custom adapters in the federated repositories configuration file.

Before you begin

Shut down the WebSphere Application Server and the wsadmin command window.

About this task

The federated repositories configuration file, wimconfig.xml, is shipped with WebSphere Application Server 6.1.x and is located in the *app_server_root*/profiles/*profile_name*/config/cells/*cell_name*/wim/config directory.

Note: For additional information about the commands to use for this topic, see “IdMgrRepositoryConfig command group for the AdminTask object” on page 835.

Use the following steps to add a custom adapter to any federated repositories configuration file and to any realm defined within the configuration file.

1. Open the wimconfig.xml file with a text editor.
2. Add a new config:repositories element to the file. This element should be placed before the config:realmConfiguration element.

   The following example configures a custom repository to use the **com.ibm.ws.wim.adapter.sample.SampleFileAdapter** class and sets the **SampleFileRepository** repository as the identifier:

   ```
   <config:repositories adapterClassName="com.ibm.ws.wim.adapter.sample.SampleFileAdapter"
   id="SampleFileRepository"/>
   ```
3. Save the wimconfig.xml file and close the text editor.
4. Copy the vmmsampleadapter.jar file that is provided to app_server_root/lib.
5. Start the wsadmin command prompt application and enter the following command:

   ```
   wsadmin –conntype none
   ```
6. Disable paging in the common repository configuration. Set the supportPaging parameter for the updateIdMgrRepository command to false to disable paging.

Note: You must perform this step because the sample adapter does not support paging.

The following examples use the **SampleFileRepository** repository as the identifier for the custom repository.

Using Jython:

```
AdminTask.updateIdMgrRepository('-id SampleFileRepository -supportPaging false')
```

Using Jacl:

```
$AdminTask updateIdMgrRepository {-id SampleFileRepository -supportPaging
false}
```

Note: A warning will appear until the configuration of the sample repository is complete.

7. Add the necessary custom properties for the adapter. Use the setIdMgrCustomProperty command repeatedly to add multiple properties. Use this command once per property to add multiple properties to your configuration. You must use both the **name** and **value** parameters to add the custom property for the specified repository. For example, to add a custom property of **fileName**, enter the following command.

   Using Jython:

   ```
   AdminTask.setIdMgrCustomProperty('-id SampleFileRepository -name fileName
   -value "c:\sampleFileRegistry.xml"')
   ```

   Using Jacl:

   ```
   $AdminTask setIdMgrCustomProperty {-id SampleFileRepository -name fileName
   -value "c:\sampleFileRegistry.xml"}
   ```

8. Add a base entry to the adapter configuration. Use the addIdMgrRepositoryBaseEntry command to specify the name of the base entry for the specified repository. For example:

   Using Jython:

   ```
   AdminTask.addIdMgrRepositoryBaseEntry('-id SampleFileRepository -name
   o=sampleFileRepository')
   ```

   Using Jacl:

   ```
   $AdminTask addIdMgrRepositoryBaseEntry {-id SampleFileRepository -name
   o=sampleFileRepository}
   ```

9. Use the addIdMgrRealmBaseEntry command to add the base entry to the realm, which will link the realm with the repository:

   Using Jython:

   ```
   AdminTask.addIdMgrRealmBaseEntry('-name defaultWIMFileBasedRealm -baseEntry o=sampleFileRepository')
   ```

   Using Jacl:

   ```
   $AdminTask addIdMgrRealmBaseEntry {-name defaultWIMFileBasedRealm -baseEntry o=sampleFileRepository}
   ```

10. Save your configuration changes. Enter the following commands to save the new configuration and close the wsadmin tool.

    Using Jython:

    ```
    AdminConfig.save()
    exit
    ```

    Using Jacl:

    ```
    $AdminConfig save
    exit
    ```

    The following example displays the complete text of the newly-revised wimconfig.xml file:

    ```
    <!--
      Begin Copyright

      Licensed Materials - Property of IBM

      virtual member manager

      (C) Copyright IBM Corp. 2005 All Rights Reserved.
    ```

```
  US Government Users Restricted Rights - Use, duplication or
  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

  End Copyright
-->
<sdo:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:config="http://www.ibm.com/websphere/wim
/config" xmlns:sdo="commonj.sdo">
  <config:configurationProvider maxPagingResults="500" maxSearchResults="4500"
maxTotalPagingResults="1000"
pagedCacheTimeOut="900" pagingEntityObject="true" searchTimeOut="600000">
    <config:dynamicModel xsdFileName="wimdatagraph.xsd"/>
    <config:supportedEntityTypes defaultParent="o=defaultWIMFileBasedRealm" name="Group">
      <config:rdnProperties>cn</config:rdnProperties>
    </config:supportedEntityTypes>
    <config:supportedEntityTypes defaultParent="o=defaultWIMFileBasedRealm" name="OrgContainer">
      <config:rdnProperties>o</config:rdnProperties>
      <config:rdnProperties>ou</config:rdnProperties>
      <config:rdnProperties>dc</config:rdnProperties>
      <config:rdnProperties>cn</config:rdnProperties>
    </config:supportedEntityTypes>
    <config:supportedEntityTypes defaultParent="o=defaultWIMFileBasedRealm" name="PersonAccount">
      <config:rdnProperties>uid</config:rdnProperties>
    </config:supportedEntityTypes>
    <config:repositories xsi:type="config:FileRepositoryType" adapterClassName="com.ibm.
    ws.wim.adapter.file.was.FileAdapter"
id="InternalFileRepository" supportPaging="false" supportSorting="false" messageDigestAlgorithm="SHA-1">
      <config:baseEntries name="o=defaultWIMFileBasedRealm"/>
    </config:repositories>
    <config:repositories adapterClassName="com.ibm.ws.wim.adapter.sample.SampleFileAdapter"
    id="SampleFileRepository">
      <config:CustomProperties name="fileName" value="c:\sampleFileRegistry.xml"/>
      <config:baseEntries name="o=sampleFileRepository"/>
    </config:repositories>
    <config:realmConfiguration defaultRealm="defaultWIMFileBasedRealm">
      <config:realms delimiter="@" name="defaultWIMFileBasedRealm" securityUse="active">
        <config:participatingBaseEntries name="o=defaultWIMFileBasedRealm"/>
        <config:participatingBaseEntries name="o=sampleFileRepository"/>
        <config:uniqueUserIdMapping propertyForInput="uniqueName" propertyForOutput="uniqueName"/>
        <config:userSecurityNameMapping propertyForInput="principalName" propertyForOutput="principalName"/>
        <config:userDisplayNameMapping propertyForInput="principalName" propertyForOutput="principalName"/>
        <config:uniqueGroupIdMapping propertyForInput="uniqueName" propertyForOutput="uniqueName"/>
        <config:groupSecurityNameMapping propertyForInput="cn" propertyForOutput="cn"/>
        <config:groupDisplayNameMapping propertyForInput="cn" propertyForOutput="cn"/>
      </config:realms>
    </config:realmConfiguration>
</config:configurationProvider></sdo:datagraph>
```

11. Restart the application server.

Related reference

Sample custom adapters for federated repositories examples
Out of the box adapters for federated repositories provide File, LDAP, and Database adapters for your use. These adapters implement the com.ibm.wsspi.wim.Repository software programming interface (SPI). A virtual member manager custom adapter needs to implement the same SPI.

Disabling embedded Tivoli Access Manager client using wsadmin

Follow these steps to unconfigure the Java Authorization Contract for Containers (JACC) provider for Tivoli Access Manager.

About this task

In a Network Deployment architecture, ensure that all the managed servers, including node agents, are started. Perform the following process once on the deployment management server. Details of the

unconfiguration are forwarded to managed servers, including node agents, when a synchronization is performed. The managed servers require their own reboot for the configuration changes to take effect.

Note: It is also possible to unconfigure using the administrative console. For details on unconfiguring the embedded Tivoli Access Manager client using the WebSphere Application Server administrative console, refer to Disabling embedded Tivoli Access Manager client using the administrative console.

1. Restart the deployment manager process.
2. Start the wsadmin command-line utility. The wsadmin command is found in the*install_dir*/bin directory
3. From the **wsadmin** prompt, enter the following command:

   ```
   WSADMIN>$AdminTask unconfigureTAM -interactive
   ```

   You are prompted to enter the following information:

Option	Description
WebSphere Application Server node name	Enter an asterisk (*) to select all nodes.
Tivoli Access Manager administrator user name	Enter the Tivoli Access Manager administration user ID, as created at the time of Tivoli Access Manager configuration. This name is usually, `sec_master`.
Tivoli Access Manager administrator user password	Enter the password for the Tivoli Access Manager administrator.
Force	Enter *yes,* if you want to ignore errors when unconfiguring the JACC provider for Tivoli Access Manager. Enter this option as *yes* only when the Tivoli Access Manager domain is in an irreparable state.
Defer	Enter `no`, to force the unconfiguration of the connected server. Enter `No` for the unconfiguration to proceed correctly.

4. When all information is entered, enter `F` to save the properties or `C` to cancel from the unconfiguration process and discard the entered information.
5. Optional: Synchronize all nodes.
6. Restart all WebSphere Application Server instances for the changes to take effect.

Configuring security auditing using scripting

Security auditing provides tracking and archiving of auditable events. This topic uses the wsadmin tool to enable and administer your security auditing configurations.

About this task

While security authentication and authorization ensures that users must have access to view protected resources, security auditing provides a mechanism to validate the integrity of a security computing environment. Security auditing collects and logs authentication, authorization, system management, security, and audit policy events in audit event records. You can analyze audit event records to determine possible security breaches, threats, attacks, and potential weaknesses in the security configuration of your environment. Enable security auditing in your environment. For example, the following list displays a sample of events to audit:

- Determine the time that a specific user attempted to access a resource.
- View information for successful and unsuccessful attempts to access resources.
- Review changes to resources that were made by a specific user.
- Determine the cause of unsuccessful login attempts.

Use the following task outline to enable and configure security auditing in your environment:

1. Enable administrative security in your environment.
2. Configure auditable events. The security auditing configuration provides four default auditable filters. Use this topic to configure filters for additional audit events.
3. Configure audit event factories. The security auditing configuration provides a default event factory. Use this topic to configure additional audit event factories.
4. Configure audit service providers. The security auditing configuration provides a default service provider. Use this topic to configure additional audit service providers.
5. Set the global audit policy. After setting up audit event factories, service providers, and events, use this topic to enable security auditing.

Results

After completing the steps to enable and configure security auditing, the profile of interest audits your security configurations for specific auditable event types.

What to do next

To further configure security auditing, you can:

- Configure audit notifications.
- Encrypt audit records.
- Sign audit records.

Configuring audit service providers using scripting

Before enabling security auditing, use this task to configure audit service providers using the wsadmin tool. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before configuring security audit service providers, enable administrative security in your environment.

About this task

In order to enable security auditing in your environment, you must configure an audit service provider. The audit service provider writes the audit records and data to the back-end repository associated with the service provide implementation. The security auditing configuration provides a default service provider. Use this topic to customize your security auditing subsystem by creating additional audit service providers.

Use the following steps to configure your security auditing subsystem using the wsadmin tool:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure an audit service provider. You can use the default binary-based audit service provider, or use this step to create a new audit service provider.

   There are binary file-based and third-party audit service providers. In addition to the default binary file-based service provider, you can configure a third-party audit service provider.

   Choose the type of audit service provider to create.

   - Use the createBinaryEmitter command and the following required parameters to create a default audit service provider:

Parameter	Description	Data Type	Required
-uniqueName	Specifies a unique name that identifies the audit service provider.	String	Yes
-className	Specifies the class implementation of the audit service provider interface.	String	Yes

Parameter	Description	Data Type	Required
-fileLocation	Specifies the file location for the audit service provider to write the audit logs.	String	Yes
-auditFilters	Specifies a reference or a group of references to predefined audit filters, using the following format: `reference, reference, reference`	String	Yes
-maxFileSize	Specifies the maximum size each audit log reaches before the system saves it with a timestamp and creates a new file. Specify the file size in megabytes. If you do not specify this parameter, the system sets the maximum file size to 10 megabytes.	Integer	No
-maxLogs	Specifies the maximum number of audit logs to create before rewriting the oldest log. If you do not specify this parameter, the system allows up to 100 audit logs before overwriting the oldest log.	Integer	No

The following example creates a new audit service provider in your security auditing configuration:

```
AdminTask.createBinaryEmitter('-uniqueName newASP -className
com.ibm.ws.security.audit.BinaryEmitterImpl -fileLocation /AUDIT_logs
-auditFilters "AuditSpecification_1173199825608, AuditSpecification_1173199825609,
AuditSpecification_1173199825610, AuditSpecification_1173199825611"')
```

- Use the createThirdPartyEmitter command to use a third-party audit service provider. Use the following parameters with the createThirdPartyEmitter command:

Parameter	Description	Data Type	Required
-uniqueName	Specifies a unique name that identifies the audit service provider.	String	Yes
-className	Specifies the class implementation of the audit service provider interface.	String	Yes
-eventFormatterClass	Specifies the class that implements how the audit event is formatted for output. If you do not specify this parameter, the system uses the standard text format for output.	String	Yes
-auditFilters	Specifies a reference identifier or a group of reference identifiers to pre-defined audit filters, using the following format: reference, reference, reference.	String	Yes
-customProperties	Specifies any custom properties that might be required to configure a third party audit service provider.	String	No

The following example creates a new third party audit service provider in your security auditing configuration:

```
AdminTask.createThirdPartyEmitter('-uniqueName myAuditServiceProvider -className
com.mycompany.myclass -fileLocation /auditLogs -auditFilters
"AuditSpecification_1173199825608, AuditSpecification_1173199825609,
AuditSpecification_1173199825610, AuditSpecification_1173199825611"')
```

3. Save your configuration changes.

What to do next

Enable security auditing in your environment.

Configuring audit event factories using scripting

Before enabling security auditing, use this task to configure audit event factories using the wsadmin tool. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before configuring security auditing event factories, enable administrative security in your environment.

About this task

In order to enable security auditing in your environment, you must configure an audit event factory. The audit event factory gathers the data that is associated with security events. The security auditing configuration provides a default event factory. Use this topic to customize your security auditing subsystem by creating additional audit event factories.

Use the following steps to configure your security auditing subsystem using the wsadmin tool:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure event filters. You can use the default event filters or use this step to create additional filters to customize your security auditing configuration.

   The application server provides the following event filters by default in the `audit.xml` template file:

Event Name	Outcome of event
SECURITY_AUTHN	SUCCESS
SECURITY_AUTHN	DENIED
SECURITY_RESOURCE_ACCESS	SUCCESS
SECURITY_AUTHN	REDIRECT

   You can configure additional audit event types to track and archive various events. Use the following command to list all supported auditable events:

```
print AdminTask.getSupportedAuditEvents()
```

   Use the createAuditFilter command with the -eventType and -outcome parameters to enable one or multiple audit events and outcomes. You can specify multiple event types and multiple outcomes separated by a comma with one command invocation. The following list describes each valid auditable event that you can specify with the -eventType parameter:

Event name	Description
SECURITY_AUTHN	Audits all authentication events
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities are involved
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation of an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might be related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that might represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web services
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including identity assertion, RunAs, and low assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity

For each audit event type, you must specify an outcome. Valid outcomes include SUCCESS, FAILURE, REDIRECT, ERROR, DENIED, WARNING, and INFO. The following command example creates an audit filter to log users who receive an error when modifying credentials:

```
AdminTask.createAuditFilter('-name uniqueFilterName -eventType
SECURITY_AUTHN_CREDS_MODIFY,SECURITY_AUTHN_DELEGATION -outcome ERROR,REDIRECT')
```

3. Create an audit event factory. You can use the default audit event factory or use this step to create a new audit event factory.

   Use the createAuditEventFactory command to create an audit event factory in your security configuration. You can use the default implementation of the audit event factory or use a third-party implementation. To configure a third-party implementation, use the optional -customProperties parameter to specify any properties necessary to configure the audit event factory implementation.

   Specify the following required parameters with the createAuditEventFactory to configure your audit event factory:

Parameter	Description	Data type	Required
-uniqueName	Specifies a unique name that identifies the audit event factory.	String	Yes
-className	Specifies the class implementation of the audit event factory interface.	String	Yes
-auditFilters	Specifies a reference or a group of references to predefined audit filters, using the following format: "reference, reference, reference"	String	Yes
-provider	Specifies a reference to a predefined audit service provider implementation.	String	Yes
-customProperties	Specifies a comma (,) separated list of custom property pairs to add to the security object in the following format: attribute=value,attribute=value	String	No

   The following sample command creates an enables an audit event factory:

```
AdminTask.createAuditEventFactory('-uniqueName eventFactory1 -className
com.ibm.ws.security.audit.AuditEventFactoryImpl -auditFilters
"AuditSpecification_1173199825608, AuditSpecification_1173199825609, AuditSpecification_1173199825610,
AuditSpecification_1173199825611" -provider newASP')
```

4. Save your configuration changes.

What to do next

Configure the audit service provider.

Configuring auditable events using scripting

Before enabling security auditing, use this task to configure event filters using the wsadmin tool. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before configuring security auditing filters, enable administrative security in your environment.

About this task

Before configuring an audit event factory and audit service provider, configure event filters. The audit service provider writes audit records to the back end repository associated with the provider implementation. The audit event factory generates security events. Event filters specify which event types and outcomes the system audits and records. Each event type has up to seven possible outcomes, including success, failure, denied, error, warning, info, and redirect. The security auditing configuration provides four default filters. Use this topic to customize your security auditing subsystem by creating additional audit event filters.

Use the following steps to configure your security auditing subsystem using the wsadmin tool:

1. Launch the wsadmin scripting tool using the Jython scripting language.

2. Configure event filters. You can use the default event filters or use this step to create additional filters to customize your security auditing configuration.

   The application server provides the following event filters by default in the `audit.xml` template file:

Event Name	Outcome of event
SECURITY_AUTHN	SUCCESS
SECURITY_AUTHN	DENIED
SECURITY_RESOURCE_ACCESS	SUCCESS
SECURITY_AUTHN	REDIRECT

   You can configure additional audit event types to capture various events. Use the following command to list all supported auditable events:

```
print AdminTask.getSupportedAuditEvents()
```

   Use the createAuditFilter command with the -name, -eventType, and -outcome parameters to enable one or multiple audit events and outcomes. You can specify multiple event types and multiple outcomes separated by a comma with one command invocation. The following list describes each valid auditable event that you can specify with the -eventType parameter:

Event name	Description
SECURITY_AUTHN	Audits all authentication events
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities are involved
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including identity assertion, RunAs, and low assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.

   For each audit event type, you must specify an outcome. Valid outcomes include SUCCESS, FAILURE, REDIRECT, ERROR, DENIED, WARNING, and INFO. The following command example creates an audit filter to log users who receive an error when modifying credentials:

```
AdminTask.createAuditFilter('-name myUniqueName -eventType SECURITY_AUTHN_CREDS_MODIFY,SECURITY_AUTHN_DELEGATION -outcome ERROR,REDIRECT')
```

3. Save your configuration changes.

What to do next

Enable security auditing in your environment.

Enabling security auditing using scripting

Use this task to enable and configure security auditing in your environment with the wsadmin tool. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before enabling security auditing, enable administrative security in your environment.

If you previously configured security auditing and do not want to modify configuration settings, use the enableAudit and disableAudit commands to start and stop security auditing. After enabling or disabling security auditing, restart the server to apply the configuration changes.

About this task

Security auditing ensures the integrity of a security computing environment. Security auditing collects and logs authentication, authorization, system management, security, and audit policy events in audit event

records. You can analyze audit event records to determine possible security breaches, threats, attacks, and potential weaknesses in the security configuration of your environment.

Use the following steps to enable and configure security auditing in your environment:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Verify that the security auditing subsystem is configured.

   To enable security auditing, you must configure event filters, an audit emitter, and an audit event factory. Event filters specify which event types the system audits and records, and the outcome of the event. The audit service provider writes the audit records to the backend repository that is associated with the implementation. The audit event factory generates security events.

   By default, the security auditing system includes one audit service provider and one audit event factory.

   The audit command groups provide several commands to query for event filters, audit emitters, event factories, and their respective configuration attributes. Use the audit command reference to use specific query commands. The following example commands query your security auditing configuration at a high level.

   - Use the getAuditFilters command to display a list of references to all audit filters defined in your configuration, as the following example demonstrates:

```
AdminTask.getAuditFilters()
```

   - Use the listAuditEmitters command to display a list of all audit emitters in your configuration, as the following example demonstrates:

```
AdminTask.listAuditEmitters()
```

   - Use the listAuditEventFactories command to display a list of all audit event factories in your configuration, as the following example demonstrates:

```
AdminTask.listAuditEventFactories()
```

3. Enable security auditing in your environment. Use the modifyAuditPolicy command to enable security auditing in your environment. Use the following optional parameters for the modifyAuditPolicy command to customize your security auditing configuration:

Parameter	Description	Data type	Required
-auditEnabled	Specifies whether to enable security auditing.	Boolean	No
-auditPolicy	Specifies the behavior of the server process if the audit subsystem fails. Valid values are: `WARN`, `NOWARN` and `FATAL`. The `WARN` setting notifies the auditor when an error occurs and ceases auditing when an error occurs in the audit sub-system, but continues to run the application server process. The `NOWARN` setting does not notify the auditor when an error occurs and ceases auditing, but continues to run the application server process. The `FATAL` setting notifies the auditor of the error and stops the application server process. By default, the command assigns the `NOWARN` setting.	String	No
-auditorId	Specifies the ID of the user to assign to the auditor role.	String	No
-auditorPwd	Specifies the password for the auditor role.	String	No
-sign	Specifies whether to sign audit records. By default, the security auditing system does not sign audit records. You must configure the signing of audit records before you can specify this parameter.	Boolean	No

Parameter	Description	Data type	Required
-encrypt	Specifies whether to encrypt audit records. By default, the security auditing system does not encrypt audit records. You must configure encryption for audit records before you can specify this parameter.	Boolean	No
-verbose	Specifies whether to capture verbose audit data. By default, the security auditing system does not capture verbose audit data.	Boolean	No
-encryptionCert	Specifies the reference ID of the certificate to use for encryption. Specify this parameter if you set the -encrypt parameter to `true`.	String	No

The following example command enables security auditing, and identifies the primary auditor by assigning a user and password.

```
AdminTask.modifyAuditPolicy('-auditEnabled true -auditorId securityAdmin -auditorPwd security4you')
```

4. Save your configuration changes.
5. Restart the server.

Results

After completing the steps to enable and configure security auditing, the profile of interest audits your security configurations for specific auditable event types.

What to do next

After you configure the audit policy for the first time, use the enableAudit and disableAudit commands to turn the security auditing system on and off. The system maintains the settings that you define with the modifyAuditPolicy command when you enable and disable the security auditing system.

Note: You must restart the server to apply the configuration changes.

Configuring security audit notifications using scripting

Configure the security auditing system to send email notifications to a distribution list, system log, or both a distribution list and a system log if a failure occurs in the audit subsystem. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before configuring a notification object in the audit.xml configuration file, verify that you set up a security auditing subsystem and configured the security auditing policy.

About this task

You can configure the security auditing system to notify a specific person or group when a failure occurs in the audit subsystem. Use the following steps to enable security auditing email notifications, set the format of notification email, and secure email:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Customize and enable security auditing email notifications.

   Use the createAuditNotification command and the following parameters to configure notifications:

Parameter	Description	Data Types	Required
-notificationName	Specifies a unique name to assign the audit notification object in the audit.xml file.	String	Yes

Parameter	Description	Data Types	Required
-logToSystemOut	Specifies whether to log the notification to the SystemOut.log file.	Boolean	Yes
-sendEmail	Specifies whether to email notifications.	Boolean	Yes
-emailList	Specifies the email address or email distribution list to email notifications. The format for this parameter is: `admin@company.com(smtp-server.mycompany.com)`	String	No
-emailFormat	Specifies whether to send the email be `HTML` or `TEXT` format.	String	No

To create the audit notification object, you must specify the -notificationName, -logToSystemOut, and -sendEmail parameters, as the following example demonstrates:

```
AdminTask.createAuditNotification('-notificationName defaultEmailNotification
-logToSystemOut true -sendEmail true -emailList administrator@mycompany.com(smtp-server.mycompany.com)
-emailFormat HTML')
```

3. Create an audit notification monitor object.

   Create an audit notification monitor object to monitor the security auditing subsystem for possible failure. Use the createAuditNotificationMonitor command and the following parameters to create a monitor object for the security auditing system:

Parameter	Description	Data Types	Required
-notificationName	Specifies a unique name to assign the audit notification object in the audit.xml file.	String	Yes
-logToSystemOut	Specifies whether to log the notification to the SystemOut.log file.	Boolean	Yes
-sendEmail	Specifies whether to email notifications.	Boolean	Yes
-emailList	Specifies the email address or email distribution list to email notifications. The format for this parameter is: `admin@company.com(smtp-server.mycompany.com)`	String	No
-emailFormat	Specifies whether to send the email be `HTML` or `TEXT` format.	String	No

   To create the audit notification monitor object, you must specify the -notificationName, -logToSystemOut, and -sendEmail parameters, as the following example demonstrates:

```
AdminTask.createAuditNotificationMonitor('-notificationName defaultEmailNotification
-logToSystemOut true -sendEmail true -emailList administrator@mycompany.com(smtp-server.mycompany.com)
-emailFormat HTML')
```

4. Save your configuration changes.

Results

The security auditing system notifies the specified recipients if a failure occurs in the security auditing system.

What to do next

Use the modifyAuditNotification command and the Audit Notification Commands command group for the AdminTask object to manage your notification configuration.

Encrypting security audit data using scripting

You can use the wsadmin tool to configure the security auditing system to encrypt security audit records. Security auditing provides tracking and archiving of auditable events.

Before you begin

Before configuring encryption, set up your security auditing subsystem. You can enable security auditing before or after completing the steps in this topic.

Verify that you have the appropriate administrative role. To complete this topic, you must have the auditor administrative role. If you are importing a certificate from a keystore that exists in the security.xml file, you must have the auditor and administrator administrative roles.

About this task

When configuring encryption, the auditor can select one of the following choices:

- Allow the application server to automatically generate a certificate or use an existing self-signed certificate generated by the auditor.
- Use an existing keystore to store this certificate, or create a new keystore to store this certificate.

Note: To ensure that there is a separation of privileges between the administrator role and the auditor role, the auditor can create a self-signed certificate outside of the application server process and maintain the private key of that certificate.

Use the following task steps to encrypt security audit data:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure encryption settings for security audit data.

   Use the createAuditEncryptionConfig command and the following parameters to create the audit encryption model to encrypt your audit records. You must specify the -enableAuditEncryption, -certAlias, and -encryptionKeyStoreRef parameters, and either the -autogenCert or -importCert parameters.

Parameter	Description	Data Type	Required
-enableAuditEncryption	Specifies whether to encrypt audit records. This parameter modifies your audit policy configuration.	Boolean	Yes
-certAlias	Specifies the alias name that identifies the generated or imported certificate.	String	Yes
-encryptionKeyStoreRef	Specifies the reference ID of the keystore to import the certificate to.	String	Yes
-autogenCert	Specifies whether to automatically generate the certificate used to encrypt the audit records. You must specify either this parameter or the -importCert parameter, but you cannot specify both.	Boolean	No
-importCert	Specifies whether to import an existing certificate to encrypt the audit records. You must specify either this parameter or the -autogenCert parameter, but you cannot specify both.	Boolean	No
-certKeyFileName	Specifies the unique name of the key file from which the certificate is imported.	String	No
-certKeyFilePath	Specifies the key file location from which the certificate is imported.	String	No

Parameter	Description	Data Type	Required
-certKeyFileType	Specifies the key file type from which the certificate is imported.	String	No
-certKeyFilePassword	Specifies the key file password from which the certificate is imported.	String	No
-certAliasToImport	Specifies the alias from which the certificate is imported.	String	No

The following command example configures encryption and supports the system to automatically generate the certificate:

```
AdminTask.createAuditEncryptionConfig('-enableAuditEncryption true -certAlias auditCertificate
-autogenCert true -encryptionKeyStoreRef auditKeyStore')
```

The following command example configures encryption and imports a certificate:

```
AdminTask.createAuditEncryptionConfig('-enableAuditEncryption true -certAlias auditCertificate
-importCert true -certKeyFileName MyServerKeyFile.p12 -certKeyFilePath
install_root/etc/MyServerKeyFile.p12 -certKeyFileType PKCS12 -certKeyFilePassword password4key
-certAliasToImport defaultCertificate -encryptionKeyStoreRef auditKeyStore')
```

3. You must restart the server to apply configuration changes.

Results

Encryption is configured for security audit data. If you set the -enableAuditEncryption parameter to `true`, then your security auditing system encrypts security audit data when security auditing is enabled.

What to do next

After you configure the encryption model for the first time, then you may use the enableAuditEncryption and disableAuditEncryption commands to turn encryption on and off.

The following example uses the enableAuditEncryption command to turn on encryption:

```
AdminTask.enableAuditEncryption()
```

The following example uses the disableAuditEncryption command to turn off encryption:

```
AdminTask.disableAuditEncryption()
```

Signing security audit data using scripting

You can use the wsadmin tool to configure the security auditing system to sign security audit records. Security auditing provides tracking and archiving of auditable events.

Before you begin

Verify that you have the appropriate administrative role. To complete this topic, you must have the auditor and administrator administrative roles.

About this task

When configuring the signing of audit data, the auditor can choose between the following options:

- Allow the application server to automatically generate a certificate.
- Use an existing self-signed certificate that the auditor previously generated.
- Use the same self-signed certificate as the system uses to encrypt the audit records.
- Use an existing keystore to store this certificate.
- Create a new keystore to store this certificate.
- Use an existing self-signed certificate in an existing keystore.

Use the following task steps to configure the signing of security audit data:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Configure signing settings for security audit data.

   Use the createAuditSigningConfig command to create the signing model to sign your audit records.

   You can import the certificate from an existing key file name that contains that certificate, automatically generate the certificate, or use the same certificate as used to encrypt the audit records. The signing keystore must exist in the `security.xml` file. The system updates this keystore with the certificate to use to sign the audit records. Use the parameters in the following table with the createAuditSigningConfig command. You must specify the -enableAuditSigning, -certAlias, and -signingKeyStoreRef parameters.

Parameter	Description	Data Type	Required
-enableAuditSigning	Specifies whether to sign audit records. This parameter modifies your audit policy configuration.	Boolean	Yes
-certAlias	Specifies the alias name that identifies the generated or imported certificate.	String	Yes
-signingKeyStoreRef	Specifies the reference ID of the keystore to import the certificate to.	String	Yes
-useEncryptionCert	Specifies whether to use the same certificate for encryption and signing. You must specify the -useEncryptionCert, -autogenCert, or -importCert parameter.	Boolean	No
-autogenCert	Specifies whether to automatically generate the certificate used to sign the audit records. You must specify the -useEncryptionCert, -autogenCert, or -importCert parameter.	Boolean	No
-importCert	Specifies whether to import an existing certificate to sign the audit records. You must specify the -useEncryptionCert, -autogenCert, or -importCert parameter.	Boolean	No
-certKeyFileName	Specifies the unique name of the key file for the certificate to import.	String	No
-certKeyFilePath	Specifies the key file location for the certificate to import.	String	No
-certKeyFileType	Specifies the key file type for the certificate to import.	String	No
-certKeyFilePassword	Specifies the key file password for the certificate to import.	String	No
-certAliasToImport	Specifies the alias of the certificate to import.	String	No

   The following command example configures signing and allows the system to automatically generate the certificate:

```
AdminTask.createAuditSigningConfig('-enableAuditSigning true -certAlias auditSigningCert
-autogenCert true -signingKeyStoreRef Ref_Id_of_KeyStoreInSecurityXML')
```

   The following command example configures signing and imports a certificate:

```
AdminTask.createAuditSigningConfig('-enableAuditSigning true -certAlias auditSigningCert
-importCert true -certKeyFileName MyServerKeyFile.p12 -certKeyFilePath install_root/etc/MyServerKeyFile.p12
-certKeyFileType PKCS12 -certKeyFilePassword password4key -certAliasToImport defaultCertificate
-signingKeyStoreRef Ref_Id_of_KeyStoreInSecurityXML')
```

   The following command example uses the same certificate for signing and encryption:

```
AdminTask.createAuditSigningConfig('-enableAuditSigning true -certAlias auditSigningCert
-useEncryptionCert true -signingKeyStoreRef Ref_Id_of_KeyStoreInSecurityXML')
```

3. Save your configuration changes.
4. Restart the server to apply the configuration changes.

Results

Signing is configured for your security audit data. If you set the -enableAuditSigning parameter to `true`, your security auditing system signs security audit data when security auditing is enabled.

What to do next

Once you configure the signing model for the first time, use the enableAuditSigning and disableAuditSigning commands to quickly turn signing on and off. The following example uses the enableAuditSigning command to turn signing on:

```
AdminTask.enableAuditSigning()
```

The following example uses the disableAuditSigning command to turn signing off:

```
AdminTask.disableAuditSigning()
```

AuditKeyStoreCommands command group for the AdminTask object

You can use the Jython scripting language to configure the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditKeyStoreCommands group to configure audit keystores in the security auditing system.

Use the following commands to manage audit key stores in the `audit.xml` configuration file:

- createAuditKeyStore
- deleteAuditKeyStore
- getAuditKeyStoreInfo
- listAuditKeyStores
- modifyAuditKeyStore

createAuditKeyStore

Creates a keystore in the `audit.xml` file. The system uses this keystore to encrypt audit records.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

-keyStoreType
Specifies a valid keystore type. The default keystore type is `PKCS12`. (String, required)

-keyStoreLocation
Specifies the location where the system creates the keystore. (String, required)

-keyStorePassword
Specifies the password for the keystore. (String, required)

-keyStorePasswordVerify
Verifies the password for the keystore. (String, required)

Optional parameters

-keyStoreProvider
Specifies a provider for the keystore. (String, optional)

-keyStoreIsFileBased
Specifies if the keystore is file-based. The default is `true`. (Boolean, optional)

-keyStoreHostList
Specifies the host list for the keystore. (String, optional)

-keyStoreInitAtStartup
Specifies whether the system initializes the keystore on startup. The default is `false`. (Boolean, optional)

-keyStoreReadOnly
Specifies whether the keystore is read-only or not. Default is `false`. (Boolean, optional)

-keyStoreStashFile
Specifies whether the keystore needs a stash file. (Boolean, optional)

-enableCryptoOperations
Specifies whether the keystore is an acceleration keystore. False default. (Boolean, optional)

-scopeName
Specifies the scope for the keystore. (String, optional)

-keyStoreDescription
Specifies a description for the keystore. (String, optional)

Return value

The command returns the ID of the new keystore, as the following example displays:

```
KeyStore_1173199825578
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditKeyStore('-keyStoreName mynewkeystore -keyStoreLocation
c:\install_root\appserver\profiles\AppSrv01\config\cells -keyStorePassword
myPwd -keyStorePasswordVerify myPwd -keyStoreProvider IBMJCE -scopeName (cell):Node04Cell')
```

- Using Jython list:

```
AdminTask.createAuditKeyStore(['-keyStoreName', 'mynewkeystore', '-keyStoreLocation',
'c:\install_root\appserver\profiles\AppSrv01\config\cells', '-keyStorePassword',
'myPwd', '-keyStorePasswordVerify', 'myPwd', '-keyStoreProvider', 'IBMJCE',
'-scopeName', '(cell):Node04Cell'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditKeyStore('-interactive')
```

deleteAuditKeyStore

The deleteAuditKeyStore command removes the reference to an audit keystore from the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the name of the keystore. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the keystore. (String, optional)

-removeKeyStoreFile
Specifies whether to remove the keystore from the configuration. Specify this parameter if the keystore of interest is not in use. (Boolean, optional)

Return value

The command returns a value of `true` if the system successfully removes the reference to the keystore from the `audit.xml` configuration file.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditKeyStore('-keyStoreName AuditDefaultKeyStore -scopeName
(cell):Node04Cell -removeKeyStoreFile false')
```

- Using Jython list:

```
AdminTask.deleteAuditKeyStore(['-keyStoreName', 'AuditDefaultKeyStore', '-scopeName',
'(cell):Node04Cell', '-removeKeyStoreFile', 'false'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditKeyStore('-interactive')
```

getAuditKeyStoreInfo

The getAuditKeyStoreInfo command returns a list of attributes for the keystore that the system uses to encrypt audit records.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name to identify the keystore. (String, required)

Optional parameters

-scopeName
Specifies the management scope of the keystore. (String, optional)

Return value

The command returns a list of attributes for the keystore, as the following sample output displays:

```
{{location ${CONFIG_ROOT}/audittrust.p12}
{password *****}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#KeyStore_1173199825578}
{_Websphere_Config_Data_Version {}}
{useForAcceleration false}
{slot 0}
{type PKCS12}
{additionalKeyStoreAttrs {}}
```

```
{fileBased true}
{_Websphere_Config_Data_Type KeyStore}
{customProviderClass {}}
{hostList {}}
{createStashFileForCMS false}
{description {keyStore description}}
{readOnly false}
{initializeAtStartup true}
{managementScope (cells/Node04Cell|audit.xml#ManagementScope_1173199825608)}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditKeyStoreInfo('-keyStoreName AuditDefaultKeyStore')
```

- Using Jython list:

```
AdminTask.getAuditKeyStoreInfo(['-keyStoreName', 'AuditDefaultKeyStore'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditKeyStoreInfo('-interactive')
```

listAuditKeyStores

The listAuditKeyStores command lists the attributes for the audit keystores within a specific management scope or for all audit keystores.

The user must have the monitor administrative role to run this command.

Target object

None.

Optional parameters

-scopeName
Specifies the management scope associated with the keystores of interest. (String, optional)

-all
Specifies whether to list all keystores. When the -all parameter is set as `true`, it overrides the -scopeName parameter. (Boolean, optional)

Return value

The command returns a list of attributes for the scope of interest, as the following sample output displays:

```
{{location ${CONFIG_ROOT}/audittrust.p12}
{password *****}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#KeyStore_1173199825578}
{_Websphere_Config_Data_Version {}}
{useForAcceleration false}
{slot 0}
{type PKCS12}
{additionalKeyStoreAttrs {}}
{fileBased true}
{_Websphere_Config_Data_Type KeyStore}
{customProviderClass {}}
{hostList {}}
{keyStoreRef KeyStore_1173199825578}
{createStashFileForCMS false}
{description {keyStore description}}
{managementScope (cells/Node04Cell|audit.xml#ManagementScope_1173199825608)}
{readOnly false}
{initializeAtStartup true}
{usage {}}
{provider IBMJCE}{name AuditDefaultKeyStore}}
{{location c:\install_root\appserver\profiles\AppSrv01\config\cells}
{password *****}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#KeyStore_1184700968484}
{_Websphere_Config_Data_Version {}}
{useForAcceleration false}
{slot 0}
```

```
{type PKCS12}
{additionalKeyStoreAttrs {}}
{fileBased true}
{_Websphere_Config_Data_Type KeyStore}
{customProviderClass {}}
{hostList {}}
{keyStoreRef KeyStore_1184700968484}
{createStashFileForCMS false}
{description {}}
{managementScope {}}
{readOnly false}
{initializeAtStartup false}
{usage {}}
{provider IBMJCE}
{name mykeystore}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditKeyStores('-scopeName (cell):Node04Cell')
```

- Using Jython list:

```
AdminTask.listAuditKeyStores(['-scopeName', '(cell):Node04Cell'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditKeyStores('-interactive')
```

modifyAuditKeyStore

The modifyAuditKeyStore command modifies the keystore reference in the `audit.xml` file. The command edits keystore that encrypts audit records.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

Optional parameters

-scopeName
Specifies the scope name of this keystore. (String, optional)

-keyStoreType
Specifies valid keystore type. (String, optional)

-keyStoreLocation
Specifies the location where the system creates the keystore. (String, optional)

-keyStorePassword
Specifies the password for this keystore. (String, optional)

-keyStoreIsFileBased
Specifies whether the keystore is file based. (Boolean, optional)

-keyStoreInitAtStartup
Specifies whether the system should initialize the keystore at startup. (Boolean, optional)

-keyStoreReadOnly
Specifies whether the keystore is read-only or editable. (Boolean, optional)

-keyStoreDescription
Specifies a description for the keystore. (String, optional)

Return value

The command returns a value of `true` if the system successfully modifies the keystore.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditKeyStore('-keyStoreName AuditDefaultKeyStore -scopeName
(cell):Node04Cell -keyStoreType PKCS12 -keyStoreLocation
c:\install_root\appserver\profiles\AppSrv01\config\cells\Node04Cell\audittrust.p12
-keyStorePassword myPwd')
```

- Using Jython list:

```
AdminTask.modifyAuditKeyStore(['-keyStoreName', 'AuditDefaultKeyStore', '-scopeName',
'(cell):Node04Cell', '-keyStoreType', 'PKCS12', '-keyStoreLocation',
'c:\install_root\appserver\profiles\AppSrv01\config\cells\Node04Cell\audittrust.p12',
'-keyStorePassword', 'myPwd'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditKeyStore('-interactive')
```

AuditEmitterCommands for the AdminTask object

You can use the Jython scripting language to configure audit service providers with the wsadmin tool. Use the commands and parameters in the AuditEmitterCommands group to create, manage, and remove audit service providers from your security auditing system configuration.

Use the following commands to configure audit service providers:

createBinaryEmitter

The createBinaryEmitter command creates an entry in the `audit.xml` file to reference the configuration of the binary file emitter implementation of the audit service provider interface.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies a name to uniquely identify this implementation of the audit service provider interface. (String, required)

-className
Specifies the class that implements the audit service provider interface. (String, required)

-fileLocation
Specifies the location where the system writes the audit logs. (String, required)

-auditFilters
Specifies a reference or a group of references to predefined audit filters. Use the following format to specify multiple references: `reference,reference,reference` (String, required)

Optional parameters

-eventFormatterClass
Specifies the class that implements how the system formats the audit event for output. If you want to use the default audit service provider, do not specify this parameter. (String, optional)

-maxFileSize
Specifies the maximum size that each log reaches before the system saves the audit log with a timestamp. Specify the size in megabytes. The default value is 10 MB. (Integer, optional)

-maxLogs
Specifies the maximum number of log files to create before the system rewrites the oldest audit log. The default value is 100 logs. (Integer, optional)

Return value

The command returns the shortened reference ID for the audit service provider, as the following sample output displays:

```
AuditServiceProvider_1184686384968
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createBinaryEmitter('-uniqueName mybinaryemitter -className
com.ibm.ws.security.audit.BinaryEmitterImpl -fileLocation
c:\wasinstall\appserver\profiles\AppSrv01\logs\server1 -maxFileSize 20 -maxLogs
100 -auditFilters AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.createBinaryEmitter(['-uniqueName', 'mybinaryemitter', '-className',
'com.ibm.ws.security.audit.BinaryEmitterImpl', '-fileLocation',
'c:\wasinstall\appserver\profiles\AppSrv01\logs\server1', '-maxFileSize',
'20', '-maxLogs', '100', '-auditFilters', 'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createBinaryEmitter('-interactive')
```

createSMFEmitter

The createSMFEmitter command creates an entry in the `audit.xml` file to reference the configuration of an SMF implementation of the audit service provider interface. The encryption and signing of audit records is not supported for SMF implementations.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies a name to uniquely identify this implementation of the audit service provider interface. (String, required)

-auditFilters
Specifies a reference or a group of references to predefined audit filters. Use the following format to specify multiple references: `reference,reference,reference` (String, required)

Return value

The command returns the shortened reference ID for the audit service provider, as the following sample output displays:

```
AuditServiceProvider_1184686384968
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createSMFEmitter('-uniqueName mySMFEmitter -auditFilters
AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.createSMFEmitter(['-uniqueName', 'mySMFEmitter', '-auditFilters',
'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createSMFEmitter('-interactive')
```

createThirdPartyEmitter

The createThirdPartyEmitter command creates an entry in the `audit.xml` configuration file to reference the configuration of a third party emitter implementation of the audit service provider interface. The encryption and signing of audit records is not supported for third party implementations.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies a name to uniquely identify this implementation of the audit service provider interface. (String, required)

-className
Specifies the class that implements the audit service provider interface. (String, required)

-auditFilters
Specifies a reference or a group of references to predefined audit filters. Use the following format to specify multiple references: `reference,reference,reference` (String, required)

Optional parameters

-eventFormatterClass
Specifies the class that implements how the system formats the audit event for output. (String, optional)

-customProperties
Specifies any custom properties that the system might need to configure the third party implementation of the audit service provider. Use the following format to specify the custom properties: `name=value,name=value` (String, optional)

Return value

The command returns the shortened reference ID to the audit service provider, as the following example output displays:

```
AuditServiceProvider_1184686638218
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createThirdPartyEmitter('-uniqueName myThirdPartyEmitter -className
com.mycompany.myemitterclass -eventFormatterClass com.mycompany.myeventformatterclass
-auditFilters AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.createThirdPartyEmitter(['-uniqueName', 'myThirdPartyEmitter', '-className',
'com.mycompany.myemitterclass', '-eventFormatterClass', 'com.mycompany.myeventformatterclass',
'-auditFilters', 'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createThirdPartyEmitter('-interactive')
```

deleteAuditEmitterByRef

The deleteAuditEmitterByRef command deletes the audit service provider implementation that the system references with the reference id. If an event factory is using the audit service provider, the system generates an error that indicates that the system cannot remove the audit service provider.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies the reference identifier of the audit service provider implementation to delete. (String, required)

Return value

The command returns a value of `true` if the system successfully removes the audit service provider.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditEmitterByRef('–emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.deleteAuditEmitterByRef(['–emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditEmitterByRef('-interactive')
```

deleteAuditEmitterByName

The deleteAuditEmitterByName command deletes the audit service provider implementation that the system references with the unique name. If an event factory is using the audit service provider, the system generates an error that indicates that the system cannot remove the audit service provider.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies the name that uniquely identifies this implementation of the audit service provider interface to delete. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit service provider implementation.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditEmitterByName('-uniqueName mybinaryemitter')
```

- Using Jython list:

```
AdminTask.deleteAuditEmitterByName(['-uniqueName', 'mybinaryemitter'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditEmitterByName('-interactive')
```

getAuditEmitter

The getAuditEmitter command returns the attributes for the audit service provider of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a audit service provider implementation. (String, required)

Return value

The command returns an attribute list for the audit service provider specified by the -emitterRef parameter, as the following example output displays:

```
{{auditSpecifications myfilter(cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1184598886859)}
{name auditServiceProviderImpl_1}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditServiceProvider_1173199825608}
{maxFileSize 1}
{_Websphere_Config_Data_Type AuditServiceProvider}
{fileLocation ${PROFILE_ROOT}/logs/server1}
{className com.ibm.ws.security.audit.BinaryEmitterImpl}
{properties {}}
{eventFormatterClass {}}
{maxLogs 100}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEmitter('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getAuditEmitter(['-emitterRef AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEmitterClass('-interactive')
```

getBinaryFileLocation

The getBinaryFileLocation command returns the file location of the binary file audit logs.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a binary file audit service provider implementation. (String, required)

Return value

The command returns the file path of the audit log, as the following example displays:

```
$profile_root/logs/server1
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getBinaryFileLocation('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getBinaryFileLocation(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getBinaryFileLocation('-interactive')
```

getAuditEmitterFilters

The getAuditEmitterFilters command returns a list of defined filters for the audit service provider implementation of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies the audit service provider implementation of interest. You can specify a reference to the service provider object. (String, required)

Return value

The command returns a list of defined filters in a shortened format, as the following example output displays:

```
AUTHN:SUCCESS,AUTHN:INFO,AUTHZ:SUCCESS,AUTHZ:INFO
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEmitterFilters('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getAuditEmitterFilters(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEmitterFilters('-interactive')
```

getBinaryFileSize

The getBinaryFileSize command returns the maximum file size of the binary audit log that is defined for the audit service provider of interest in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a binary file audit service provider implementation. (String, required)

Return value

The command returns the integer value of the maximum file size in megabytes.

Batch mode example usage

- Using Jython string:

```
AdminTask.getBinaryFileSize('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getBinaryFileSize(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getBinaryFileSize('-interactive')
```

getEmitterClass

The getEmitterClass command returns the class name of the audit service provider emitter implementation.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a audit service provider implementation. (String, required)

Return value

The command returns the class name of the audit service provider implementation.

Batch mode example usage

- Using Jython string:

```
AdminTask.getEmitterClass('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getEmitterClass(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEmitterClass('-interactive')
```

getEmitterUniqueId

The getEmitterUniqueId command returns the unique identifier of the audit service provider implementation.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a service provider implementation. (String, required)

Return value

The command returns the unique ID of the audit service provider of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getEmitterUniqueId('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getEmitterUniqueId(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEmitterUniqueId('-interactive')
```

getMaxNumBinaryLogs

The getMaxNumBinaryLogs command returns the maximum number of binary audit logs that is defined for the audit service provider of interest in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a binary file audit service provider implementation. (String, required)

Return value

The command returns the integer value that represents the maximum number of binary audit logs in the configuration.

Batch mode example usage

- Using Jython string:

```
AdminTaskgetMaxNumBinaryLogs('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTaskgetMaxNumBinaryLogs(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getMaxNumBinaryLogs('-interactive')
```

listAuditEmitters

The listAuditEmitters command returns a list of configured audit service provider implementation objects and the corresponding attributes.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns an array list of audit service provider implementation objects and attributes, as the following example output displays:

```
{{auditSpecifications myfilter(cells/CHEYENNENode04Cell|audit.xml#AuditSpecifica
tion_1184598886859)}
{name auditServiceProviderImpl_1}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditServiceProvid
er_1173199825608}
{maxFileSize 1}
{_Websphere_Config_Data_Type AuditServiceProvider}
{fileLocation ${PROFILE_ROOT}/logs/server1}
```

```
{className com.ibm.ws.security.audit.BinaryEmitterImpl}
{properties {}}
{auditSpecRef1 AuditSpecification_1184598886859}
{eventFormatterClass {}}
{maxLogs 100}
{emitterRef AuditServiceProvider_1173199825608}}
{{auditSpecifications DefaultAuditSpecification_1(cells/CHEYENNENode04Cell|audit
.xml#AuditSpecification_1173199825608)}
{name mythirdpartyemitter}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditServiceProvid
er_1184686638218}
{maxFileSize 0}
{_Websphere_Config_Data_Type AuditServiceProvider}
{fileLocation {}}
{className com.mycompany.myemitterclass}
{properties {}}
{auditSpecRef1 AuditSpecification_1173199825608}
{eventFormatterClass com.mycompany.myeventformatterclass}
{maxLogs 0}
{emitterRef AuditServiceProvider_1184686638218}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditEmitters()
```

- Using Jython list:

```
AdminTask.listAuditEmitters()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditEmitters('-interactive')
```

modifyAuditEmitter

The modifyAuditEmitter command modifies the attributes of an audit service provider implementation object.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a audit service provider implementation. (String, required)

-auditFilters
Specifies a reference or a group of references to predefined audit filters. Use the following format to specify multiple references: `reference,reference,reference` (String, required)

-fileLocation
Specifies the location where the system writes the audit logs. (String, required)

Optional parameters

-className
Specifies the class name to use to identify the implementation. (String, optional)

-eventFormatterClass
Specifies the class that implements how the system formats the audit event for output. If you want to use the default audit service provider, do not specify this parameter. (String, optional)

-customProperties
Specifies a list of custom properties formatted as name and value pairs in the following format: `name=value,name=value`. (String, optional)

-maxFileSize
Specifies the maximum size that each log reaches before the system saves the audit log with a timestamp. Specify the size in megabytes. The default value is 10 MB. (Integer, optional)

-maxLogs
Specifies the maximum number of log files to create before the system rewrites the oldest audit log. The default value is 100 logs. (Integer, optional)

Return value

The command returns a value of `true` if the system successfully modifies the audit service provider of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditEmitter('-emitterRef AuditServiceProvider_1184686638218
-auditFilters AuditSpecification_1173199825608
-fileLocation c:\wasinstall\appserver\profiles\AppSrv01\mylogs -maxFileSize
14 -maxLogs 200')
```

- Using Jython list:

```
AdminTask.modifyAuditEmitter(['-emitterRef', 'AuditServiceProvider_1184686638218',
'-auditFilters', 'AuditSpecification_1173199825608', '-fileLocation',
'c:\wasinstall\appserver\profiles\AppSrv01\mylogs', '-maxFileSize', '14', '-maxLogs',
'200'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditEmitter('-interactive')
```

setAuditEmitterFilters

The setAuditEmitterFilters command sets the filters for an audit service provider implementation.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies a reference to a audit service provider implementation. (String, required)

-filtersRef
Specifies one or more references to defined audit filters. Use the following format to specify more than one filter reference: `reference,reference,reference` (String, required)

Return value

The command returns a value of `true` if the system successfully sets the filters for the audit service provider.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAuditEmitterFilters('-emitterRef AuditServiceProvider_1173199825608
-filtersRef AuditSpecification_1184598886859')
```

- Using Jython list:

```
AdminTask.setAuditEmitterFilters(['-emitterRef', 'AuditServiceProvider_1173199825608',
'-filtersRef', 'AuditSpecification_1184598886859'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setAuditEmitterFilters('-interactive')
```

AuditSigningCommands command group for the AdminTask object

You can use the Jython scripting language to configure the signing of audit records with the wsadmin tool. Use the commands and parameters in the AuditSigningCommands group to enable, disable, and configure the security audit system to sign audit records.

Use the following commands to configure audit signing:

- "createAuditSigningConfig"
- "deleteAuditSigningConfig" on page 761
- "disableAuditSigning" on page 762
- "enableAuditSigning" on page 762
- "getAuditSigningConfig" on page 763
- "importEncryptionCertificate" on page 763
- "isAuditSigningEnabled" on page 764
- "modifyAuditSigningConfig" on page 765

createAuditSigningConfig

The createAuditSigningConfig command creates the signing model that the system uses to sign the audit records. Use this command to configure your audit signing configuration for the first time. If you have already configured audit signing, use the enableAuditSigning and disableAuditSigning commands to turn audit signing on and off.

You can import the certificate from an existing key file name containing that certificate, automatically generate the certificate, or use the same certificate as the application server uses encrypt the audit records. To use an existing certificate in an existing keystore, specify input values for the -enableAuditEncryption, -certAlias, and –signingKeyStoreRef parameters. Also, set the value of the -useEncryptionCert, –autogenCert, and –importCert parameters as `false` for this scenario.

The user must have the administrator and auditor administrative roles to run this command.

Target object

None.

Required parameters

-enableAuditSigning
Specifies whether to sign audit records. This parameter modifies your audit policy configuration. (Boolean, required)

-certAlias
Specifies the alias name that identifies the generated or imported certificate. (String, required)

-signingKeyStoreRef
Specifies the reference ID of the key store that system imports the certificate to. The signing keystore must already exist in the security.xml file. The system updates this keystore with the certificate that is used to sign the audit records. (String, required)

Optional parameters

-useEncryptionCert
Specifies whether to use the same certificate for encryption and signing. (Boolean, optional)

-autogenCert
Specifies whether to automatically generate the certificate used to sign the audit records. (Boolean, optional)

-importCert
Specifies whether to import an existing certificate to sign the audit records. (Boolean, optional)

-certKeyFileName
Specifies the unique name of the key file for the certificate to import. (String, optional)

-certKeyFilePath
Specifies the key file location for the certificate to import. (String, optional)

-certKeyFileType
Specifies the key file type for the certificate to import. (String, optional)

-certKeyFilePassword
Specifies the key file password for the certificate to import. (String, optional)

-certAliasToImport
Specifies the alias of the certificate to import. (String, optional)

Return value

If successful, returns the shortened from of the keystore where the signing certificate has been added to. Remember, this keystore is in the security.xml file, not the audit.xml file.

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditSigningConfig('-enableAuditSigning true -certAlias
auditSigningCert -autogenCert true -signingKeyStoreRef Ref_Id_of_KeyStoreInSecurityXML')
```

- Using Jython list:

```
AdminTask.createAuditSigningConfig(['-enableAuditSigning', 'true', '-certAlias',
'auditSigningCert', '-autogenCert', 'true -signingKeyStoreRef',
'Ref_Id_of_KeyStoreInSecurityXML'])
```

Interactive mode example usage

- Using Jython :

```
AdminTask.createAuditSigningConfig('-interactive')
```

deleteAuditSigningConfig

The deleteAuditSigningConfig command deletes the signing model that the system uses to sign the audit records. When the system deletes the audit signing configuration, it does not delete the key store file in the security.xml or the signer certificate for the keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully removes the audit signing configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditSigningConfig()
```

- Using Jython list:

```
AdminTask.deleteAuditSigningConfig()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditSigningConfig('-interactive')
```

disableAuditSigning

The disableAuditSigning command disables audit record signing for the security auditing system.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully disables audit signing.

Batch mode example usage

- Using Jython string:

```
AdminTask.disableAuditSigning()
```

- Using Jython list:

```
AdminTask.disableAuditSigning()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.disableAuditSigning('-interactive')
```

enableAuditSigning

The enableAuditSigning command enables audit record signing in the security auditing system.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully enables audit signing in the security auditing system.

Batch mode example usage

- Using Jython string:

```
AdminTask.enableAuditSigning()
```

- Using Jython list:

```
AdminTask.enableAuditSigning()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.enableAuditSigning()
```

getAuditSigningConfig

The getAuditSigningConfig command retrieves the signing model that the system uses to sign the audit records.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes that are associated with the signing model, as the following sample output displays:

```
{{securityXmlSignerScopeName (cell):Node04Cell:(node):Node04}
{securityXmlSignerCertAlias mysigningcert}
{securityXmlSignerKeyStoreName NodeDefaultRootStore}
{signerKeyStoreRef KeyStore_Node04_4}
{enabled true}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditSigningConfig()
```

- Using Jython list:

```
AdminTask.getAuditSigningConfig()
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.getAuditSigningConfig('-interactive')
```

importEncryptionCertificate

The importEncryptionCertificate command imports the self-signed certificate used for encrypting audit data from the encryption keystore into another keystore. Use this command internally to automatically generate a certificate for either encryption or signing. You can also use this command to import the certificate into the keystore by specifying the keyStoreName and keyStoreScope parameters.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name to identify the keystore. (String, required)

-keyFilePath
Specifies the keystore path name that contains the certificate to import. (String, required)

-keyFilePassword
Specifies the password of the keystore that contains the certificate to import. (String, required)

-keyFileType
Specifies the type of the keystore. (String, required)

-certificateAliasFromKeyFile
Specifies the alias of the certificate to import from the keystore file. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-certificateAlias
Specifies a unique name to identify the imported certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully imports the encryption certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.importEncryptionCertificate('{-keyStoreName AuditDefaultKeyStore -keyStoreScope
(cell):Node04Cell -keyFilePath c:/install_root/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12
-keyFilePassword WebAS -keyFileType PKCS12 -certificateAliasFromKeyFile root -certificateAlias myimportcert')
```

- Using Jython list:

```
AdminTask.importEncryptionCertificate(['{-keyStoreName', 'AuditDefaultKeyStore', '-keyStoreScope',
'(cell):Node04Cell', '-keyFilePath', 'c:/install_root/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12',
'-keyFilePassword', 'WebAS', '-keyFileType', 'PKCS12', '-certificateAliasFromKeyFile',
'root', '-certificateAlias', 'myimportcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importEncryptionCertificate('-interactive')
```

isAuditSigningEnabled

The isAuditSigningEnabled command indicates whether audit signing is enabled or disable in the security audit system.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if signing is configured in the security auditing system.

Batch mode example usage

- Using Jython string:

```
AdminTask.isAuditSigningEnabled()
```

- Using Jython list:

```
AdminTask.isAuditSigningEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isAuditSigningEnabled('-interactive')
```

modifyAuditSigningConfig

The modifyAuditSigningConfig command modifies the signing model that the system uses to sign the audit records.

The certificate may either be imported from an existing key file name containing that certificate, automatically generated, or be the same certificate used to encrypt the audit records. To use an existing certificate in an existing keystore, specify input values for the -enableAuditEncryption, -certAlias, and –signingKeyStoreRef parameters. Also, set the value the -useEncryptionCert, –autogenCert, and –importCert parameters as `false` for this scenario.

The user must have the administrator and auditor administrative roles to run this command.

Target object

None.

Required parameters

-enableAuditSigning
Specifies whether to sign audit records. This parameter modifies your audit policy configuration. (Boolean, required)

-certAlias
Specifies the alias name that identifies the generated or imported certificate. (String, required)

-signingKeyStoreRef
Specifies the reference ID of the key store that system imports the certificate to. The signing keystore must already exist in the security.xml file. The system updates this keystore with the certificate that is used to sign the audit records. (String, required)

Optional parameters

-useEncryptionCert
Specifies whether to use the same certificate for encryption and signing. (Boolean, optional)

-autogenCert
Specifies whether to automatically generate the certificate used to sign the audit records. (Boolean, optional)

-importCert
Specifies whether to import an existing certificate to sign the audit records. (Boolean, optional)

-certKeyFileName
Specifies the unique name of the key file for the certificate to import. (String, optional)

-certKeyFilePath
Specifies the key file location for the certificate to import. (String, optional)

-certKeyFileType
Specifies the key file type for the certificate to import. (String, optional)

-certKeyFilePassword
Specifies the key file password for the certificate to import. (String, optional)

-certAliasToImport
Specifies the alias of the certificate to import. (String, optional)

Return value

The command returns a value of `true` if the system successfully modifies the security auditing system configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditSigningConfig('-enableAuditSigning true -certAlias auditSigningCert
-autogenCert true -signingKeyStoreRef Ref_Id_of_KeyStoreInSecurityXML')
```

- Using Jython list:

```
AdminTask.modifyAuditSigningConfig(['-enableAuditSigning', 'true', '-certAlias',
'auditSigningCert', '-autogenCert', 'true', '-signingKeyStoreRef', 'Ref_Id_of_KeyStoreInSecurityXML'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditSigningConfig('-interactive')
```

AuditEncryptionCommands command group for the AdminTask object

You can use the Jython scripting language to configure the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditEncryptionCommands group to configure the security audit system to encrypt audit records.

Use the following commands to enable, disable, and configure audit record encryption:

createAuditEncryptionConfig

The createAuditEncryptionConfig command creates the encryption model used to encrypt the audit records.

You can import the certificate from an existing key file name containing that certificate or automatically generate a certificate.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-enableAuditEncryption
Specifies whether to encrypt audit records. This parameter modifies your audit policy configuration. (Boolean, required)

-certAlias
Specifies the alias name that identifies the generated or imported certificate. (String, required)

-encryptionKeyStoreRef
Specifies the reference ID of the keystore to import the certificate to. (String, required)

Optional parameters

-autogenCert
Specifies whether to automatically generate the certificate used to encrypt the audit records. You must specify either this parameter or the -importCert parameter, but you cannot specify both. (Boolean, optional)

-importCert
Specifies whether to import an existing certificate to encrypt the audit records. You must specify either this parameter or the -autogenCert parameter, but you cannot specify both. (Boolean, optional)

-certKeyFileName
Specifies the unique name of the key file for the certificate to import. (String, optional)

-certKeyFilePath
Specifies the key file location for the certificate to import. (String, optional)

-certKeyFileType
Specifies the key file type for the certificate to import. (String, optional)

-certKeyFilePassword
Specifies the key file password for the certificate to import. (String, optional)

-certAliasToImport
Specifies the alias of the certificate to import. (String, optional)

Return value

The command returns the shortened form of the reference ID of the created encryption keystore if the system successfully creates the audit encryption configuration, as the following example output displays:

```
KeyStore_1173199825578
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditEncryptionConfig('-enableAuditEncryption true -certAlias
auditCertificate -autogenCert true -encryptionKeyStoreRef auditKeyStore')
```

- Using Jython list:

```
AdminTask.createAuditEncryptionConfig(['-enableAuditEncryption', 'true', '-certAlias',
'auditCertificate', '-autogenCert', 'true', '-encryptionKeyStoreRef', 'auditKeyStore'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.createAuditEncryptionConfig('-interactive')
```

createAuditSelfSignedCertificate

The createAuditSelfSignedCertificate command creates a self-signed certificate. Use this command internally to automatically generate a certificate for encryption and signing or to import that certificate into the keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore where the system imports the self-signed certificate to. (String, optional)

-certificateAlias
Specifies a unique alias name for the certificate. (String, required)

-certificateSize
Specifies the size that the private key uses for the personal certificate. The default value is `1024`. (Integer, required)

-certificateCommonName
Specifies the common name portion of the distinguished name. (String, required)

Optional parameters

-certificateOrganization
Specifies the organizational part of the distinguished name. (String, optional)

-keyStoreScope
Specifies the scope of the keystore that the system imports the self-signed certificate to. (String, optional)

-certificateVersion
Specifies the version of the personal certificate. (String, optional)

-certificateOrganizationalUnit
Specifies the organization unit part of the distinguished name. (String, optional)

-certificateLocality
Specifies the locality portion of the distinguished name. (String, optional)

-certificateState
Specifies the state portion of the distinguished name. (String, optional)

-certificateZip
Specifies the zip code portion of the distinguished name. (String, optional)

-certificateCountry
Specifies the country portion of the distinguished name. The default value is `US`. (String, optional)

-certificateValidDays
Specifies the length of time, in days, which the certificate is valid. The default value is `365` days. (Integer, optional)

Return value

The command returns a value of `true` if the system successfully creates the self-signed certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditSelfSignedCertificate('-keyStoreName AuditDefaultKeyStore -keyStoreScope
(cell):Node04Cell -certificateAlias myNew -certificateCommonName cn=oet -certificateOrganization mycompany')
```

- Using Jython list:

```
AdminTask.createAuditSelfSignedCertificate(['-keyStoreName', 'AuditDefaultKeyStore', '-keyStoreScope',
'(cell):Node04Cell', '-certificateAlias', 'myNew', '-certificateCommonName', 'cn=oet',
'-certificateOrganization', 'mycompany'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditSelfSignedCertificate('-interactive')
```

deleteAuditCertificate

The deleteAuditCertificate command deletes a self-signed certificate from an audit keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore from which the system deletes the self-signed certificate. (String, required)

-certificateAlias
Specifies a unique alias name for the certificate to delete. (String, required)

Optional parameters

-keyStoreScope
Specifies a unique alias name for the certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully deletes the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditCertificate('-keyStoreName myKeystore -certificateAlias oldCertificate')
```

- Using Jython list:

```
AdminTask.deleteAuditCertificate(['-keyStoreName', 'myKeystore', '-certificateAlias', 'oldCertificate'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditCertificate('-interactive')
```

deleteAuditEncryptionConfig

The deleteAuditEncryptionConfig command deletes the encryption model used to encrypt the audit records. The command does not remove keystore files or the certificates.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully deletes the audit encryption configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditEncryptionConfig()
```

- Using Jython list:

```
AdminTask.deleteAuditEncryptionConfig()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditEncryptionConfig('-interactive')
```

disableAuditEncryption

The disableAuditEncryption command disables the encryption of audit records.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully disables audit record encryption.

Batch mode example usage

- Using Jython string:

```
AdminTask.disableAuditEncryption()
```

- Using Jython list:

```
AdminTask.disableAuditEncryption()
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.disableAuditEncryption('-interactive')
```

enableAuditEncryption

The enableAuditEncryption command enables the encryption of audit records.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully enables audit record encyption.

Batch mode example usage

- Using Jython string:

```
AdminTask.enableAuditEncryption()
```

- Using Jython list:

```
AdminTask.enableAuditEncryption()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.enableAuditEncryption()
```

exportAuditCertificate

The exportAuditCertificate command exports a self-signed certificate from a keystore. To use this command, you must adhere to the following user role and privilege guidelines:

- You must have audit privileges to export the certificate from an audit keystore.
- You must have the auditor and administrator roles to export the certificate to a security keystore.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

-keyStorePassword
Specifies the password that the system uses to access the keystore specified with the -keyStoreName parameter. (String, required)

-keyFilePath
Specifies the key store path name that contains the certificate to export. (String, required)

-keyFilePassword
Specifies the password of the keystore that contains the certificate to export. (String, required)

-keyFileType
Specifies the type of the keystore. (String, required)

-certificateAlias
Specifies the alias of the certificate to export from the keystore. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-aliasInKeyStore
Specifies a new unique name to identify the exported certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully exports the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportAuditCertificate('-keyStoreName AuditDefaultKeyStore -keyStoreScope
(cell):Node04Cell -keyFilePath c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12
-keyFilePassword myPwd -keyFileType PKCS12 -certificateAlias root')
```

- Using Jython list:

```
AdminTask.exportAuditCertificate(['-keyStoreName', 'AuditDefaultKeyStore', '-keyStoreScope',
'(cell):Node04Cell', '-keyFilePath', 'c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12',
'-keyFilePassword', 'myPwd', '-keyFileType', 'PKCS12', '-certificateAlias', 'root'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportAuditCertificate('-interactive')
```

exportAuditCertToManagedKS

The exportAuditCertToManagedKS command exports a self-signed certificate from an audit keystore to a managed audit keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the managed keystore. (String, required)

-keyStorePassword
Specifies the password of the managed keystore that contains the certificate to export. (String, required)

-toKeyStoreName
Specifies the unique name of the managed keystore that contains the certificate to export. (String, required)

-certificateAlias
Specifies a unique name to identify the exported certificate. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-toKeyStoreScope
Specifies the scope of the managed keystore that contains the certificate to export. (String, optional)

-aliasInKeyStore
Specifies the new unique name to identify the exported certificate. If you do not specify a value for this parameter, the system sets the unique name to the value specified for the -certificateAlias parameter. (String, optional)

Return value

The command returns a value of `true` if the system successfully exports the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportAuditCertToManagedKS('-keyStoreName  auditEncryptionKeyStore -keyStorePassword myPwd
-toKeyStoreName  AuditTrustStore -toKeyStoreScope (cell):my03Cell -certificateAlias newauditcert
-aliasInKeyStore newauditcert1')
```

- Using Jython list:

```
AdminTask.exportAuditCertToManagedKS(['-keyStoreName', 'auditEncryptionKeyStore', '-keyStorePassword', 'myPwd',
'-toKeyStoreName', 'AuditTrustStore', '-toKeyStoreScope', '(cell):my03Cell', '-certificateAlias', 'newauditcert',
'-aliasInKeyStore', 'newauditcert1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportAuditCertToManagedKS('-interactive')
```

getAuditCertificate

The getAuditCertificate command retrieves the attributes for an audit self-signed certificate in an audit keystore.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the managed keystore of interest. (String, required)

-certificateAlias
Specifies a unique name to identify the exported certificate of interest. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore of interest. (String, optional)

Return value

The command returns a list of attributes associated with the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditCertificate('-keyStoreName auditEncryptionKeyStore -certificateAlias
newauditcert')
```

- Using Jython list:

```
AdminTask.getAuditCertificate(['-keyStoreName', 'auditEncryptionKeyStore', '-certificateAlias',
'newauditcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditCertificate('-interactive')
```

getAuditEncryptionConfig

The getAuditEncryptionConfig command retrieves the encryption model that the system uses to encrypt the audit records.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes associated with the encryption model, as the following example output displays:

```
{{certRef Certificate_1184698729015}
{keystoreRef KeyStore_1173199825578}
{keyStore AuditDefaultKeyStore(cells/CHEYENNENode04Cell|audit.xml#KeyStore_1173199825578)}
{enabled true}
{alias mycertalias}
{_Websphere_Config_Data_Version {}}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#Certificate_1184698729015}
{_Websphere_Config_Data_Type Certificate}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEncryptionConfig()
```

- Using Jython list:

```
AdminTask.getAuditEncryptionConfig()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEncryptionConfig('-interactive')
```

getEncryptionKeyStore

The getEncryptionKeyStore command retrieves the attributes for the keystore that contains the certificate that the system uses to encrypt the audit records.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes for the keystore of interest, as the following example displays:

```
{{location ${CONFIG_ROOT}/audittrust.p12}
{password *****}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#KeyStore_1173199825578}
{_Websphere_Config_Data_Version {}}
{useForAcceleration false}
{slot 0}
{type PKCS12}
{additionalKeyStoreAttrs {}}
{fileBased true}
{_Websphere_Config_Data_Type KeyStore}
{customProviderClass {}}
{hostList {}}
{keystoreRef KeyStore_1173199825578}
{createStashFileForCMS false}
{description {keyStore description}}
{managementScope (cells/CHEYENNENode04Cell|audit.xml#ManagementScope_1173199825608)}
{readOnly false}
{initializeAtStartup true}
{usage {}}
{provider IBMJCE}
{name AuditDefaultKeyStore}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getEncryptionKeyStore()
```

- Using Jython list:

```
AdminTask.getEncryptionKeyStore()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEncryptionKeyStore('-interactive')
```

importAuditCertFromManagedKS

The importAuditCertFromManagedKS command imports a self-signed certificate into a keystore from a managed audit keystore. Use this command internally to automatically generate a certificate for encryption or signing and to import a certificate into the keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the managed keystore. (String, required)

-fromKeyStoreName
Specifies the unique name of the managed keystore that contains the certificate to import. (String, required)

-fromKeyStorePassword
Specifies the password of the managed keystore that contains the certificate to import. (String, required)

-certificateAliasFromKeyFile
Specifies the alias of the certificate to import from the managed keystore file. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-fromKeyStoreScope
Specifies the scope of the managed keystore that contains the certificate to import. (String, optional)

-certificateAlias
Specifies a unique name to identify the imported certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully imports the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.importAuditCertFromManagedKS('-keyStoreName AuditDefaultKeyStore -keyStoreScope
(cell):myNode03Cell -fromKeyStoreName AuditSecondDefaultKeyStore -fromKeyStoreScope
(cell):myNode03Cell -fromKeyStorePassword myPwd
-certificateAliasFromKeyFile root -certificateAlias myimportcert')
```

- Using Jython list:

```
AdminTask.importAuditCertFromManagedKS(['-keyStoreName', 'AuditDefaultKeyStore', '-keyStoreScope',
'(cell):Node04Cell', '-fromKeyStoreName', 'AuditSecondDefaultKeyStore', '-fromKeyStoreScope',
'(cell):myNode03Cell', '-fromKeyStorePassword', 'myPwd', '-certificateAliasFromKeyFile',
'root', '-certificateAlias', 'myimportcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importAuditCertFromManagedKS('-interactive')
```

importAuditCertificate

The importAuditCertificate command imports a self-signed certificate into a keystore. Use this command internally to automatically generate a certificate for encryption or signing and to import a certificate into the keystore. To use this command, you must adhere to the following user role and privilege guidelines:

- You must have audit privileges to import the certificate to an audit keystore.
- You must have the auditor and administrator roles to import the certificate to a security keystore.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

-keyFilePath
Specifies the key store path name that contains the certificate to import. (String, required)

-keyFilePassword
Specifies the password of the keystore that contains the certificate to import. (String, required)

-keyFileType
Specifies the type of the keystore. (String, required)

-certificateAliasFromKeyFile
Specifies the alias of the certificate to import from the keystore file. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-certificateAlias
Specifies a unique name to identify the imported certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully imports the audit certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.importAuditCertificate('-keyStoreName AuditDefaultKeyStore -keyStoreScope
(cell):Node04Cell -keyFilePath c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12
-keyFilePassword myPwd -keyFileType PKCS12 -certificateAliasFromKeyFile root -certificateAlias myimportcert')
```

- Using Jython list:

```
AdminTask.importAuditCertificate(['-keyStoreName', 'AuditDefaultKeyStore', '-keyStoreScope', '(cell):Node04Cell',
'-keyFilePath', 'c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12',
'-keyFilePassword', 'myPwd', '-keyFileType', 'PKCS12', '-certificateAliasFromKeyFile', 'root',
'-certificateAlias', 'myimportcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importAuditCertificate('-interactive')
```

importEncryptionCertificate

The importEncryptionCertificate command imports the self-signed certificate that the system uses to encrypt audit data from the encryption keystore into a managed keystore in security.xml.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

-keyFilePath
Specifies the key store path name that contains the certificate to import. (String, required)

-keyFilePassword
Specifies the password of the keystore that contains the certificate to import. (String, required)

-keyFileType
Specifies the type of the keystore. (String, required)

-certificateAliasFromKeyFile
Specifies the alias of the certificate to import from the keystore file. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore. (String, optional)

-certificateAlias
Specifies a unique name to identify the imported certificate. (String, optional)

Return value

The command returns a value of `true` if the system successfully imports the encryption certificate.

Batch mode example usage

- Using Jython string:

```
AdminTask.importEncryptionCertificate('-keyStoreName DefaultKeyStore -keyStoreScope (cell):Node04Cell
-keyFilePath c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12
-keyFilePassword myPwd -keyFileType PKCS12 -certificateAliasFromKeyFile root -certificateAlias myimportcert')
```

- Using Jython list:

```
AdminTask.importEncryptionCertificate(['-keyStoreName', 'DefaultKeyStore', '-keyStoreScope', '(cell):Node04Cell',
'-keyFilePath', 'c:/wasinstall/appserver/profiles/AppSrv01/config/cells/Node04Cell/nodes/Node04/trust.p12',
'-keyFilePassword', 'myPwd', '-keyFileType', 'PKCS12', '-certificateAliasFromKeyFile', 'root',
'-certificateAlias', 'myimportcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importEncryptionCertificate('-interactive')
```

isAuditEncryptionEnabled

The isAuditEncryptionEnabled command determines if audit record encryption is enabled.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if audit record encryption is enabled.

Batch mode example usage

- Using Jython string:

```
AdminTask.isAuditEncryptionEnabled()
```

- Using Jython list:

```
AdminTask.isAuditEncryptionEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isAuditEncryptionEnabled('-interactive')
```

listAuditEncryptionKeyStores

The listAuditEncryptionKeyStores command retrieves the attributes for each configured encryption keystore from the `audit.xml` file. The command returns attributes for active and inactive keystores.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes for each configured keystore, as the following example output displays:

```
{{location ${CONFIG_ROOT}/audittrust.p12}
{password *****}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#KeyStore_1173199825578}
{useForAcceleration false}
{slot 0}
{type PKCS12}
{additionalKeyStoreAttrs {}}
{fileBased true}
{_Websphere_Config_Data_Type KeyStore}
{customProviderClass {}}
{hostList {}}
{keystoreRef KeyStore_1173199825578}
{createStashFileForCMS false}
{description {keyStore description}}
{readOnly false}
{initializeAtStartup true}
{managementScope (cells/CHEYENNENode04Cell|audit.xml#ManagementScope_1173199825608)}
{usage {}}
{provider IBMJCE}
{name AuditDefaultKeyStore}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditEncryptionKeyStores()
```

- Using Jython list:

```
AdminTask.listAuditEncryptionKeyStores()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditEncryptionKeyStores('-interactive')
```

listCertAliases

The listCertAliases command retrieves a list of the personal certificates in the keystore, as specified by the keystore name and scope of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-keyStoreName
Specifies the unique name of the keystore. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope of the keystore. The default value is the cell scope. (String, optional)

Return value

The command returns a list of certificate aliases for the personal certificates that are configured for the keystore, as the following sample output displays:

```
mycertalias
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listCertAliases('-keyStoreName AuditDefaultKeyStore -keyStoreScope (cell):Node04Cell')
```

- Using Jython list:

```
AdminTask.listCertAliases(['-keyStoreName AuditDefaultKeyStore -keyStoreScope (cell):Node04Cell'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listCertAliases('-interactive')
```

modifyAuditEncryptionConfig

The modifyAuditEncryptionConfig command modifies the encryption model that the system uses to encrypt the audit records. Specify values for the -enableAuditEncryption, -certAlias, and encryptionKeyStoreRef parameters to use an existing keystore. Do not specify the -importCert or -autogenCert parameters if you use an existing keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

None

Optional parameters

-enableAuditEncryption
Specifies whether to encrypt audit records. This parameter modifies your audit policy configuration. (Boolean, optional)

-autogenCert
Specifies whether to automatically generate the certificate used to encrypt the audit records. You must specify either this parameter or the -importCert parameter, but you cannot specify both. (Boolean, optional)

-importCert
Specifies whether to import an existing certificate to encrypt the audit records. You must specify either this parameter or the -autogenCert parameter, but you cannot specify both. (Boolean, optional)

-certKeyFileName
Specifies the unique name of the key file for the certificate to import. (String, optional)

-certKeyFilePath
Specifies the key file location for the certificate to import. (String, optional)

-certKeyFileType
Specifies the key file type for the certificate to import. (String, optional)

-certKeyFilePassword
Specifies the key file password for the certificate to import. (String, optional)

-certAliasToImport
Specifies the alias of the certificate to import. (String, optional)

-certAlias
Specifies the alias name that identifies the generated or imported certificate. (String, optional)

-encryptionKeyStoreRef
Specifies the reference ID of the keystore to import the certificate to. (String, optional)

Return value

The command returns a value of `true` if the system successfully updates the configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditEncryptionConfig('-enableAuditEncryption true -certAlias mycertalias
-encryptionKeyStoreRef KeyStore_1173199825578')
```

- Using Jython list:

```
AdminTask.modifyAuditEncryptionConfig(['-enableAuditEncryption', 'true', '-certAlias', 'mycertalias',
'-encryptionKeyStoreRef', 'KeyStore_1173199825578'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditEncryptionConfig('-interactive')
```

renewAuditCertificate

The renewAuditCertificate command renews a self signed certificate in an audit keystore.

The user must have the auditor administrative role to run this command.

Target object

None.

-keyStoreName
Specifies the unique name of the managed keystore of interest. (String, required)

-certificateAlias
Specifies a unique name to identify the exported certificate to renew. (String, required)

Optional parameters

-keyStoreScope
Specifies the scope name of the keystore of interest. (String, optional)

Return value

The command returns a value of `true` if the system successfully updates the configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.renewAuditCertificate('-keyStoreName auditEncryptionKeyStore
-certificateAlias newauditcert')
```

- Using Jython list:

```
AdminTask.renewAuditCertificate(['-keyStoreName', 'auditEncryptionKeyStore',
'-certificateAlias', 'newauditcert'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.renewAuditCertificate('-interactive')
```

AuditEventFactoryCommands for the AdminTask object

You can use the Jython scripting language to configure the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditEventFactoryCommands group to configure the default or a third-party audit event factory.

Use the following commands to configure the default audit event factory or a third-party audit event factory:

createAuditEventFactory

The createAuditEventFactory command creates an audit event factory in your security auditing system configuration. You can use the default implementation of the audit event factory or use a third-party implementation. To configure a third-party implementation, use the optional -customProperties parameter to specify any properties necessary to configure the audit event factory implementation.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies a unique name that identifies the audit event factory. (String, required)

-className
Specifies the class implementation of the audit event factory interface. (String, required)

-provider
Specifies a reference to a predefined audit service provider implementation. (String, required)

-auditFilters
Specifies a reference or a group of references to predefined audit filters, using the following format: `reference, reference, reference` (String, required)

Optional parameters

-customProperties
Specifies any custom properties necessary to configure a third-party implementation. (String, optional)

Return value

The command returns the shortened reference ID for the newly created audit event factory.

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditEventFactory('-uniqueName myeventfactory -className
com.mycompany.myeventfactoryclass -provider AuditServiceProvider_1173199825608
-customProperties a=b -auditFilters AuditSpecification_1184598886859')
```

- Using Jython list:

```
AdminTask.createAuditEventFactory(['-uniqueName', 'myeventfactory', '-className',
'com.mycompany.myeventfactoryclass', '-provider', 'AuditServiceProvider_1173199825608',
'-customProperties', 'a=b', '-auditFilters', 'AuditSpecification_1184598886859'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditEventFactory()
```

deleteAuditEventFactoryByName

The deleteAuditEventFactoryByName command deletes the audit event factory implementation in the `audit.xml` file that matches a specific unique name identifier.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-uniqueName
Specifies the unique name of the audit event factory implementation. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit event factory.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditEventFactoryByName('–uniqueName
myeventfactory')
```

- Using Jython list:

```
AdminTask.deleteAuditEventFactoryByName(['–uniqueName', 'myeventfactory'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditEventFactoryByName('-interactive')
```

deleteAuditEventFactoryByRef

The deleteAuditEventFactoryByRef command deletes the audit event factory implementation that matches the reference ID of interest.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit event factory.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditEventFactoryByRef('–eventFactoryRef
AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.deleteAuditEventFactoryByRef(['–eventFactoryRef', 'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditEventFactoryByRef('–interactive')
```

getAuditEventFactory

The getAuditEventFactory command retrieves the list of attributes for the audit event factory implementation in the `audit.xml` file for a specific reference id.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns an attribute list for the audit event factory implementation of interest, as the following example output displays:

```
{{name myeventfactory}
{properties {{{validationExpression {}}
{name a}
{description {}}
{value b}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#Property_1184688593531}
{_Websphere_Config_Data_Type Property}
{required false}}}}
{className com.mycompany.myeventfactoryclass}
{auditServiceProvider auditServiceProviderImpl_1(cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608)}
{auditSpecifications DefaultAuditSpecification_1(cells/Node04Cell|audit.xml#AuditSpecification_1173199825608)}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditEventFactory_1184688293515}
{_Websphere_Config_Data_Type AuditEventFactory}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEventFactory('-eventFactoryRef AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.getAuditEventFactory(['-eventFactoryRef', 'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEventFactory('-interactive')
```

getAuditEventFactoryClass

The getAuditEventFactoryClass command retrieves the class name of the audit event factory implementation that matches a specific reference ID in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns the class name of the audit event factory of interest, as the following sample output displays:

```
com.mycompany.myeventfactoryclass
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEventFactoryClass('-eventFactoryRef
AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.getAuditEventFactoryClass(['-eventFactoryRef',
'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEventFactoryClass('-interactive')
```

getAuditEventFactoryFilters

The getAuditEventFactoryFilters command retrieves a list of defined filters for the passed-in event factory.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns a list of the defined filters for the event factory reference of interest in a shortened format, as the following sample output displays:

```
AUTHN:SUCCESS,AUTHN:INFO,AUTHZ:SUCCESS,AUTHZ:INFO
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEventFactoryFilters('-eventFactoryRef
AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.getAuditEventFactoryFilters(['-eventFactoryRef',
'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEventFactoryFilters('-interactive')
```

getAuditEventFactoryName

The getAuditEventFactoryName command retrieves the unique name of the audit event factory implementation that matches a specific reference ID in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns the name of the audit event factory, as the following sample output displays:

```
myeventfactory
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEventFactoryName('-eventFactoryRef
AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.getAuditEventFactoryName(['-eventFactoryRef',
'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEventFactoryName('-interactive')
```

getAuditEventFactoryProvider

The getAuditEventFactoryProvider command retrieves the object name of the audit service provider that a specific audit event factory implementation uses in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Return value

The command returns the object name of the audit service provider for the audit event factory of interest, as the following sample output displays:

```
auditServiceProviderImpl_1(cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608)
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditEventFactoryProvider('-eventFactoryRef
AuditEventFactory_1184688293515')
```

- Using Jython list:

```
AdminTask.getAuditEventFactoryProvider(['-eventFactoryRef',
'AuditEventFactory_1184688293515'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditEventFactoryProvider('-interactive')
```

listAuditEventFactories

The listAuditEventFactories command retrieves a list of audit event factory objects and their attributes that are defined in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns an array list of audit event factories and attributes, as the following example output displays:

```
{{auditSpecifications DefaultAuditSpecification_1(cells/Node04Cell|audit.xml#AuditSpecification_1173199825608)
DefaultAuditSpecification_2(cells/Node04Cell|audit.xml#AuditSpecification_1173199825609)
DefaultAuditSpecification_3(cells/Node04Cell|audit.xml#AuditSpecification_1173199825610)
DefaultAuditSpecification_4(cells/Node04Cell|audit.xml#AuditSpecification_1173199825611)}
{name auditEventFactoryImpl_1}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditEventFactory_1173199825608}
{_Websphere_Config_Data_Type AuditEventFactory}
{auditSpecRef4 AuditSpecification_1173199825611}
{properties {}}
{auditSpecRef3 AuditSpecification_1173199825610}
{className com.ibm.ws.security.audit.AuditEventFactoryImpl}
{auditServiceProvider auditServiceProviderImpl_1(cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608)}
{auditSpecRef2 AuditSpecification_1173199825609}
{auditSpecRef1 AuditSpecification_1173199825608}
{auditEventFactoryRef AuditEventFactory_1173199825608}
{emitterRef AuditServiceProvider_1173199825608}}
{{auditSpecifications myfilter(cells/Node04Cell|audit.xml#AuditSpecification_1184598886859)}
{name myeventfactory}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditEventFactory_1184688293515}
{_Websphere_Config_Data_Type AuditEventFactory}
{className com.mycompany.myeventfactoryclass}
{auditServiceProvider auditServiceProviderImpl_1(cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608)}
{properties {{{validationExpression {}}
{name a}
{description {}}
{value b}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#Property_1184688293546}
{_Websphere_Config_Data_Type Property}
{required false}}}}
{auditSpecRef1 AuditSpecification_1184598886859}
{auditEventFactoryRef AuditEventFactory_1184688293515}
{emitterRef AuditServiceProvider_1173199825608}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditEventFactories()
```

- Using Jython list:

```
AdminTask.listAuditEventFactories()
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.listAuditEventFactories('-interactive')
```

modifyAuditEventFactory

The modifyAuditEventFactory command modifies the attributes of the audit event factory implementation that the command references with the reference id.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

Optional parameters

-provider
Specifies the reference ID of an audit service provider implementation. (String, optional)

-className
Specifies the name of the class that implements the audit event factory interface. (String, optional)

-customProperties
Specifies one or more custom properties to associate with the audit event factory of interest. Use the following format: `name=value, name=value` (String, optional)

-auditFilters
Specifies a list of references to audit filters that exist in your configuration. You can separate each item in the list with a comma (`,`), a semicolon (`;`), or a space. (String, optional)

Return value

The command returns a value of `true` if the system successfully updates the security auditing system configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditEventFactory('-eventFactoryRef AuditEventFactory_1184688293515 -provider
AuditServiceProvider_1173199825608 -customProperties b=c')
```

- Using Jython list:

```
AdminTask.modifyAuditEventFactory(['-eventFactoryRef', 'AuditEventFactory_1184688293515',
'-provider', 'AuditServiceProvider_1173199825608', '-customProperties', 'b=c'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditEventFactory('-interactive')
```

setAuditEventFactoryFilters

The setAuditEventFactoryFilters command sets the filters for an audit event factory implementation.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-eventFactoryRef
Specifies an audit event factory implementation. This parameter can be a reference to the event factory object. (String, required)

-filtersRef
Specifies a list of references to defined audit filters. (String, required)

Return value

The command returns a value of `true` if the system successfully sets the filters for the audit event factory.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAuditEventFactoryFilters('-eventFactoryRef AuditEventFactory_1184688293515
-filtersRef AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.setAuditEventFactoryFilters(['-eventFactoryRef', 'AuditEventFactory_1184688293515',
'-filtersRef', 'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.setAuditEventFactoryFilters('-interactive')
```

AuditFilterCommands command group for the AdminTask object

You can use the Jython scripting language to configure the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditFilterCommands group to configure and manage auditable events.

Use the following commands to configure filters for auditable events in your security auditing configuration:

convertFilterRefToString

The convertFilterRefToString command converts a reference ID of a filter to a shortened string value such as `AUTHN:SUCCESS`.

Target object

None.

Required parameters

-filterRef
Specifies a reference ID for a specific audit filter in the `audit.xml` file. The system defines 4 default audit filters by default. Use the createAuditFilter command to create additional audit filters in your `audit.xml` configuration file. (String, required)

Return value

The command returns the string value of an event type in a shortened format, as the following sample output displays:

```
AUTHN:SUCCESS,AUTHN:INFO,AUTHZ:SUCCESS,AUTHZ:INFO
```

Batch mode example usage

- Using Jython string:

```
AdminTask.convertFilterRefToString('-filterRef AuditSpecification_1184598886859')
```

- Using Jython list:

```
AdminTask.convertFilterRefToString(['-filterRef', 'AuditSpecification_1184598886859'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.convertFilterRefToString('-interactive')
```

convertFilterStringToRef

The convertFilterStringToRef command converts the shortened name of an event type, such as `AUTHN:SUCCESS`, to the reference ID of the audit filter in the `audit.xml` configuration file.

The command accepts one event and outcome pair. The command does not accept multiple event and outcome pairs, such as `AUTHN:SUCCESS AUTHZ:SUCCESS`.

Target object

None.

Required parameters

-filter
Specifies a shortened form of a reference ID for an audit filter, such as `AUTHN:SUCCESS`. The event type must exist in your security auditing system configuration. (String, required)

Return value

The command returns the reference ID for the event type of interest, as the following example displays:

```
AuditSpecification_1173199825608
```

Batch mode example usage

- Using Jython string:

```
AdminTask.convertFilterStringToRef('-filter AUTHN:SUCCESS')
```

- Using Jython list:

```
AdminTask.convertFilterStringToRef(['-filter', 'AUTHN:SUCCESS'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.convertFilterStringToRef('-interactive')
```

createAuditFilter

The createAuditFilter command creates and enables a new audit event filter specification entry in the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-name
Specifies a unique name to associate with the audit event filter. (String, required)

-eventType
Specifies a list of one or more auditable events. To specify a list, separate each outcome with a comma (`,`) character. (String, required)

You can configure the following auditable events in your security auditing system:

Event name	Description
SECURITY_AUTHN	Audits all authentication events.
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities exist.
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out.
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies.
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists.
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that could represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates.
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table.
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web Services.
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services.
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including Identity Assertion, RunAs, and Low Assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity.

-outcome
Specifies a list of one or multiple event outcomes. For each audit event type, you must specify an outcome. Valid outcomes include `SUCCESS`, `FAILURE`, `REDIRECT`, `ERROR`, `DENIED`, `WARNING`, and `INFO`. (String, required)

Return value

If the system is successful, the command returns the reference ID for the new audit event filter, as the following sample output displays:

```
AuditSpecification_1184689433421
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditFilter('-name myfilter -eventType
"SECURITY_MGMT_PROVISIONING, SECURITY_MGMT_POLICY" -outcome SUCCESS')
```

- Using Jython list:

```
AdminTask.createAuditFilter(['-name', 'myfilter', '-eventType',
'"SECURITY_MGMT_PROVISIONING,', 'SECURITY_MGMT_POLICY"', '-outcome', 'SUCCESS'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditFilter('-interactive')
```

deleteAuditFilter

The deleteAuditFilter command deletes the audit event filter specification from the `audit.xml` file that the system references by an event type and outcome.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-eventType

Specifies the auditable event to delete. (String, required)

The following table displays all valid event types:

Event name	Description
SECURITY_AUTHN	Audits all authentication events.
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities exist.
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out.
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies.
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists.
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that could represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates.
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table.
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web Services.

Event name	Description
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services.
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including Identity Assertion, RunAs, and Low Assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity.

-outcome
Specifies the event outcome. Valid outcomes include `SUCCESS`, `FAILURE`, `REDIRECT`, `ERROR`, `DENIED`, `WARNING`, and `INFO`. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit filter from your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditFilter('-eventType SECURITY_AUTHN –outcome SUCCESS')
```

- Using Jython list:

```
AdminTask.deleteAuditFilter(['-eventType', 'SECURITY_AUTHN', '–outcome', 'SUCCESS'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditFilter('-interactive')
```

deleteAuditFilterByRef

The deleteAuditFilterByRef command deletes the audit filter that the system references by the referenced id.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-filterRef
Specifies the reference ID for an audit filter in your security auditing system configuration. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit filter specification from the `audit.xml` file.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditFilterByRef('-filterRef AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.deleteAuditFilterByRef(['-filterRef', 'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditFilterByRef('-interactive')
```

disableAuditFilter

The disableAuditFilter command disables the audit filter specification that corresponds to a specific reference id.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-filterRef
Specifies a reference ID for a specific audit filter in the `audit.xml` file. The system defines 4 default audit filters by default. Use the createAuditFilter command to create additional audit filters in your `audit.xml` configuration file. (String, required)

Return value

The command returns a value of `true` if the system successfully disables the audit filter.

Batch mode example usage

- Using Jython string:

```
AdminTask.disableAuditFilter('-filterRef', 'AuditSpecification_1184689433421')
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.disableAuditFilter('-interactive')
```

enableAuditFilter

The enableAuditFilter command enables the audit filter specification that corresponds to a specific reference id. Use this command to enable a filter that was previously configured and disabled in your security auditing system configuration. To create a new audit filter specification, use the creatAuditFilter command.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-filterRef
Specifies a reference ID for a specific audit filter in the `audit.xml` file. The system defines 4 default audit filters by default. Use the createAuditFilter command to create additional audit filters in your `audit.xml` configuration file. (String, required)

Return value

The command returns a value of `true` if the system successfully enables the audit filter.

Batch mode example usage

- Using Jython string:

```
AdminTask.enableAuditFilter('-filterRef AuditSpecification_1184689433421')
```

- Using Jython list:

```
AdminTask.enableAuditFilter(['-filterRef', 'AuditSpecification_1184689433421'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.enableAuditFilter('-interactive')
```

getAuditFilter

The getAuditFilter command retrieves the attributes that the system associates with the audit filter specification of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-reference
Specifies the reference ID for an audit filter in your security auditing system configuration. (String, required)

Return value

The command returns a list of attributes for the audit filter specification of interest, as the following sample output displays:

```
{{enabled true}
{name DefaultAuditSpecification_1}
{event SECURITY_AUTHN
SECURITY_AUTHN_MAPPING}
{outcome FAILURE}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditSpecification_1173199825608}
{_Websphere_Config_Data_Type AuditSpecification}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditFilter('-reference AuditSpecification_1173199825608')
```

- Using Jython list:

```
AdminTask.getAuditFilter(['-reference', 'AuditSpecification_1173199825608'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.getAuditFilter('-interactive')
```

getAuditOutcomes

The getAuditOutcomes command retrieves a list of the enabled outcomes for the auditable event type of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventType

Specifies a list of one or more auditable events. To specify a list, separate each outcome with a comma (**,**) character. (String, required)

You can retrieve the event outcome for any of the following auditable events that might be configured in your security auditing system:

Event name	Description
SECURITY_AUTHN	Audits all authentication events.
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities exist.
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out.
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies.
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists.
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that could represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates.
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table.
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web Services.
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services.
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including Identity Assertion, RunAs, and Low Assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity.

Return value

The command returns one or multiple outcomes for the event type of interest, as the following sample output displays:

```
SUCCESS
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditOutcomes('-eventType SECURITY_MGMT_PROVISIONING')
```

- Using Jython list:

```
AdminTask.getAuditOutcomes(['-eventType', 'SECURITY_MGMT_PROVISIONING'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.getAuditOutcomes('-interactive')
```

getSupportedAuditEvents

The getSupportedAuditEvents command returns a list of each supported auditable event.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns the following list of possible event outcomes:

```
SECURITY_AUTHN SECURITY_AUTHN_CREDS_MODIFY
SECURITY_AUTHN_DELEGATION SECURITY_AUTHN_MAPPING
SECURITY_AUTHN_TERMINATE SECURITY_AUTHZ
SECURITY_ENCRYPTION SECURITY_MGMT_AUDIT
SECURITY_MGMT_CONFIG SECURITY_MGMT_KEY
SECURITY_MGMT_POLICY SECURITY_MGMT_PROVISIONING
SECURITY_MGMT_REGISTRY SECURITY_MGMT_RESOURCE
SECURITY_RESOURCE_ACCESS SECURITY_RUNTIME
SECURITY_RUNTIME_KEY SECURITY_SIGNING
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getSupportedAuditEvents()
```

- Using Jython list:

```
AdminTask.getSupportedAuditEvents()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getSupportedAuditEvents('-interactive')
```

getSupportedAuditOutcomes

The getSupportedAuditOutcomes command retrieves a list of each supported outcome for the auditable event filters.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns the following list of possible event outcomes:

```
SUCCESS
INFO
WARNING
ERROR
DENIED
REDIRECT
FAILURE
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getSupportedAuditOutcomes()
```

- Using Jython list:

```
AdminTask.getSupportedAuditOutcomes()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getSupportedAuditOutcomes('-interactive')
```

isAuditFilterEnabled

The isAuditFilterEnabled command determines if the audit filter of interest is enabled in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventType

Specifies a list of one or more auditable events. To specify a list, separate each outcome with a comma (**,**) character. (String, required)

The following auditable events might be configured in your security auditing system:

Event name	Description
SECURITY_AUTHN	Audits all authentication events.
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities exist.
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out.
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies.
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists.
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that could represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates.
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table.
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web Services.
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services.
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including Identity Assertion, RunAs, and Low Assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity.

-outcome

Specifies the event outcome. Valid outcomes include `SUCCESS`, `FAILURE`, `REDIRECT`, `ERROR`, `DENIED`, `WARNING`, and `INFO`. (String, required)

Return value

The command returns a value of `true` if the event type of interest is enabled in your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.isAuditFilterEnabled('-eventType SECURITY_MGMT_PROVISIONING
-outcome SUCCESS')
```

- Using Jython list:

```
AdminTask.isAuditFilterEnabled(['-eventType', 'SECURITY_MGMT_PROVISIONING',
'-outcome', 'SUCCESS'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.isAuditFilterEnabled('-interactive')
```

isEventEnabled

The isEventEnabled command determines if the system enabled at least one audit outcome for the event of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-eventType

Specifies a list of one or more auditable events. To specify a list, separate each outcome with a comma (**,**) character. (String, required)

The following auditable events are available to configure in your security auditing system:

Event name	Description
SECURITY_AUTHN	Audits all authentication events.
SECURITY_AUTHN_MAPPING	Audits events that record mapping of credentials where two user identities exist.
SECURITY_AUTHN_TERMINATE	Audits authentication termination events such as a timeout, terminated session, or user-initiated logging out.
SECURITY_AUTHZ	Audits events related to authorization checks when the system enforces access control policies.
SECURITY_MGMT_POLICY	Audits events related to security policies, such as the creation of access control lists.
SECURITY_MGMT_PROVISIONING	Audits provisioning events such as the creation an account for a user on a specific machine or adding a user to a group on a specific machine. A given provisioning event might related to one or more SECURITY_MGMT_REGISTRY events.
SECURITY_MGMT_RESOURCE	Audits resource management events such as creation, deletion, and changes to the attributes of a resource. The resource represents an entity with operations that need to be secured. An example of a resource is the Tivoli Access Manager protected object that could represent a file, a Web page, and so on.
SECURITY_RUNTIME_KEY	Audits events related to runtime operations for certificates such as expiration, expiration checks, and invalid certificates.
SECURITY_MGMT_KEY	Audits events related to management operations for certificates such as creating, updating, or exporting a certificate, reading or updating a certificate request, publishing a certificate revocation list, monitoring changes to the keystore, truststore, and so on.
SECURITY_MGMT_AUDIT	Audits events that record operations related to the audit subsystem such as starting audit, stopping audit, turning audit on or off, changing configuration of audit filters or level, archiving audit data, purging audit data, and so on.
SECURITY_RESOURCE_ACCESS	Audits events that record all accesses to a resource. Examples are all accesses to a file, all HTTP requests and responses to a given Web page, and all accesses to a critical database table.
SECURITY_SIGNING	Audits events that record signing such as signing operations used to validate parts of a SOAP Message for Web Services.
SECURITY_ENCRYPTION	Audits events that record encryption information such as encryption for Web services.

Event name	Description
SECURITY_AUTHN_DELEGATION	Audits events that record delegation, including Identity Assertion, RunAs, and Low Assertion. Used when the client identity is propagated or when delegation involves the use of a special identity. This event type is also used when switching user identities within a given session.
SECURITY_AUTHN_CREDS_MODIFY	Audits events to modify credentials for a given user identity.

Return value

The command returns a value of `true` if the audit filter of interest has at least one outcome configured in the `audit.xml` file.

Batch mode example usage

- Using Jython string:

```
AdminTask.isEventEnabled('-eventType SECURITY_AUTHN')
```

- Using Jython list:

```
AdminTask.isEventEnabled(['-eventType', 'SECURITY_AUTHN'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isEventEnabled('-interactive')
```

listAuditFilters

The listAuditFilters command lists each audit filter and the corresponding attributes that the system defines in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of audit filters and the corresponding attributes, as the following example displays:

```
{{enabled true}
{name DefaultAuditSpecification_1}
{event SECURITY_AUTHN SECURITY_AUTHN_MAPPING}
{outcome FAILURE}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1173199825608}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1173199825608}}
{{enabled true}
{name DefaultAuditSpecification_2}
{event {}}
{outcome FAILURE}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1173199825609}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1173199825609}}
{{enabled true}
{name DefaultAuditSpecification_3}
{event SECURITY_RESOURCE_ACCESS}
{outcome FAILURE}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1173199825610}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1173199825610}}
{{enabled true}
{name DefaultAuditSpecification_4}
{event SECURITY_AUTHN_TERMINATE}
{outcome FAILURE}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1173199825611}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1173199825611}}
{{enabled true}
{name myfilter}
```

```
{event SECURITY_AUTHZ}
{outcome REDIRECT}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1184365235250}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1184365235250}}
{{enabled true}
{name myfilter1}
{event SECURITY_AUTHZ SECURITY_RESOURCE_ACCESS}
{outcome REDIRECT INFO}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1184365353218}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1184365353218}}
{{enabled true}
{name myfilter}
{event SECURITY_AUTHN SECURITY_AUTHZ}
{outcome SUCCESS INFO}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1184598886859}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1184598886859}}
{{enabled false}
{name myfilter}
{event SECURITY_MGMT_PROVISIONING SECURITY_MGMT_POLICY}
{outcome SUCCESS}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#AuditSpecification_1184689433421}
{_Websphere_Config_Data_Type AuditSpecification}
{filterRef AuditSpecification_1184689433421}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditFilters()
```

- Using Jython list:

```
AdminTask.listAuditFilters()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditFilters('-interactive')
```

listAuditFiltersByEvent

The listAuditFiltersByEvent command retrieves a list of events and event outcomes for each audit filter that is configured in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of events and event outcomes for the audit filters of interest, as the following sample output displays:

```
{AuditSpecification_1173199825608 SECURITY_AUTHN:FAILURE}{AuditSpecification_117
3199825608 SECURITY_AUTHN_MAPPING:FAILURE}{AuditSpecification_1173199825610 SECU
RITY_RESOURCE_ACCESS:FAILURE}{AuditSpecification_1173199825611 SECURITY_AUTHN_TE
RMINATE:FAILURE}{AuditSpecification_1184365235250 SECURITY_AUTHZ:REDIRECT}{Audit
Specification_1184365353218 SECURITY_AUTHZ:REDIRECT;SECURITY_AUTHZ:INFO}{AuditSp
ecification_1184365353218 SECURITY_RESOURCE_ACCESS:REDIRECT;SECURITY_RESOURCE_AC
CESS:INFO}{AuditSpecification_1184598886859 SECURITY_AUTHN:SUCCESS;SECURITY_AUTH
N:INFO}{AuditSpecification_1184598886859 SECURITY_AUTHZ:SUCCESS;SECURITY_AUTHZ:I
NFO}{AuditSpecification_1184689433421 SECURITY_MGMT_PROVISIONING:SUCCESS}{AuditS
pecification_1184689433421 SECURITY_MGMT_POLICY:SUCCESS}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditFiltersByEvent()
```

- Using Jython list:

```
AdminTask.listAuditFiltersByEvent()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditFiltersByEvent('-interactive')
```

listAuditFiltersByRef

The listAuditFiltersByRef command lists all reference ids that correspond to the audit filters that are defined in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of each reference that exists in the audit.xml configuration file, as the following sample output displays:

```
AuditSpecification_1173199825608 AuditSpecification_1173199825609
AuditSpecification_1173199825610 AuditSpecification_1173199825611
AuditSpecification_1184365235250 AuditSpecification_1184365353218
AuditSpecification_1184598886859 AuditSpecification_1184689433421
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditFiltersByRef()
```

- Using Jython list:

```
AdminTask.listAuditFiltersByRef()
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.listAuditFiltersByRef('-interactive')
```

modifyAuditFilter

The modifyAuditFilter command modifies the audit filter specification in the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-filterRef
Specifies the reference ID for the audit filter to modify in your security auditing system configuration. (String, required)

Optional parameters

-name
Specifies a unique name to associate with the audit event filter. (String, optional)

-eventType
Specifies a comma-separated list of one or more event types. (String, optional)

-outcome
Specifies a comma-separated list of one or multiple event outcomes. For each audit event type, you must specify an outcome. Valid outcomes include `SUCCESS`, `FAILURE`, `REDIRECT`, `ERROR`, `DENIED`, `WARNING`, and `INFO`. (String, optional)

-enableFilter
Specifies whether to enable the filter. Specify `true` to enable the filter, or `false` to disable the filter. (Boolean, optional).

Return value

The command returns a value of `true` if the system successfully updates the audit filter.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditFilter('-filterRef AuditSpecification_1173199825608 -name
myname -eventType SECURITY_AUTHN -outcome SUCCESS -enableFilter true')
```

- Using Jython list:

```
AdminTask.modifyAuditFilter(['-filterRef', 'AuditSpecification_1173199825608', '-name',
'myname', '-eventType', 'SECURITY_AUTHN', '-outcome', 'SUCCESS', '-enableFilter',
'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditFilter('-interactive')
```

AuditNotificationCommands command group for the AdminTask object

You can use the Jython scripting language to manage the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditNotificationCommands group to configure and manage audit notifications and audit notification monitors.

Use the following commands to configure your security auditing system notifications:

createAuditNotification

The createAuditNotification command creates an audit notification object in the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-notificationName
Specifies the unique name to assign the audit notification object. (String, required)

-logToSystemOut
Specifies whether the system logs notifications to the `SystemOut.log` file.(Boolean, required)

-sendEmail
Specifies whether to email security auditing subsystem failure notifications. (Boolean, required)

Optional parameters

-emailList
Specifies the email list to send security auditing subsystem failure notifications. (String, optional)

-emailFormat
Specifies the email format. Specify `HTML` for HTML format or `TEXT` for text format. (String, optional)

Return value

The command returns the shortened reference ID of the new audit notification object, as the following sample output displays:

```
WSNotification_1184690835390
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditNotification('[-notificationName mynotification -logToSystemOut
true -sendEmail true -emailList admin@mycompany.com(smtp-server.mycompany.com)
-emailFormat HTML]')
```

- Using Jython list:

```
AdminTask.createAuditNotification(['-notificationName', 'mynotification', '-logToSystemOut',
'true', '-sendEmail', 'true', '-emailList', 'admin@mycompany.com(smtp-server.mycompany.com)',
'-emailFormat', 'HTML'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditNotification('-interactive')
```

createAuditNotificationMonitor

The createAuditNotificationMonitor command creates an audit notification monitor object for the security auditing system. This object monitors the security auditing subsystem for possible failure.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-monitorName
Specifies the unique name of the audit notification monitoring object. (String, required)

-notificationRef
Specifies the reference ID of the audit notification object. (String, required)

-enable
Specifies whether to enable the audit notification monitor. (Boolean, required)

Return value

The command returns the shortened form of the reference ID for the audit notification monitor, as the following sample output displays:

```
AuditNotificationMonitor_1184695615171
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createAuditNotificationMonitor('-monitorName mymonitor -notificationRef
WSNotification_1184690835390 -enable true')
```

- Using Jython list:

```
AdminTask.createAuditNotificationMonitor(['-monitorName', 'mymonitor', '-notificationRef',
'WSNotification_1184690835390', '-enable', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createAuditNotificationMonitor('-interactive')
```

deleteAuditNotification

The deleteAuditNotification command deletes an audit notification object from the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-notificationRef
Specifies the reference ID of the audit notification object to delete. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit notification object from the `audit.xml` configuration file.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditNotification('-notificationRef WSNotification_1184690835390')
```

- Using Jython list:

```
AdminTask.deleteAuditNotification(['-notificationRef', 'WSNotification_1184690835390'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditNotification('-interactive')
```

deleteAuditNotificationMonitorByName

The deleteAuditNotificationMonitorByName command deletes the audit notification monitor that the user specifies with the unique name.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-monitorName
Specifies the unique name of the audit notification monitor to delete. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit notification monitor from the configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditNotificationMonitor('-monitorName mymonitor')
```

- Using Jython list:

```
AdminTask.deleteAuditNotificationMonitor(['-monitorName', 'mymonitor'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditNotificationMonitor('-interactive')
```

deleteAuditNotificationMonitorByRef

The deleteAuditNotificationMonitorByRef command deletes the audit notification monitor that the user specifies with the reference ID.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-monitorRef
Specifies the reference ID of the audit notification monitor object to delete. (String, required)

Return value

The command returns a value of `true` if the system successfully deletes the audit notification monitor of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAuditNotificationMonitor('-monitorRef
AuditNotificationMonitor_1184695615171')
```

- Using Jython list:

```
AdminTask.deleteAuditNotificationMonitor(['-monitorRef',
'AuditNotificationMonitor_1184695615171'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAuditNotificationMonitor('-interactive')
```

getAuditNotification

The getAuditNotification command retrieves the attributes for an audit notification object of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-notificationRef
Specifies the reference ID of the audit notification object of interest. (String, required)

Return value

The command returns a list of attributes for the specific audit notification object, as the following sample output displays:

```
{{name mynotification}
{sslConfig {}}
{logToSystemOut true}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#WSNotification_1184690835390}
{emailList sweetshadow@us.ibm.com(smtp-server.us.ibm.com)}
{sendEmail true}
{_Websphere_Config_Data_Type WSNotification}
{properties {}}
{emailFormat HTML}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditNotification('-notificationRef WSNotification_1184690835390')
```

- Using Jython list:

```
AdminTask.getAuditNotification(['-notificationRef', 'WSNotification_1184690835390'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditNotification('-interactive')
```

getAuditNotificationMonitor

The getAuditNotificationMonitor command retrieves the attributes that the system associates with the audit notification monitor of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-monitorRef
Specifies the reference ID of the audit notification monitor of interest. (String, required)

Return value

The command returns a list of attributes for the audit notification monitor of interest, as the following sample output displays:

```
{{name mymonitor}
{enabled true}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditNotificationMonitor_1184695615171}
{_Websphere_Config_Data_Type AuditNotificationMonitor}
{wsNotification mynotification(cells/Node04Cell|audit.xml#WSNotification_1184690835390)}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditNotificationMonitor('-monitorRef
AuditNotificationMonitor_1184695615171')
```

- Using Jython list:

```
AdminTask.getAuditNotificationMonitor(['-monitorRef',
'AuditNotificationMonitor_1184695615171'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditNotificationMonitor('-interactive')
```

getEmailList

The getEmailList command retrieves the email distribution list for the audit notification object. If the notification monitor is not configured, the audit notification object is not active and the command returns a null value.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns this email list for the active audit notification object, as the following sample output displays:

```
admin@mycompany.com(smtp-server.mycompany.com)
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getEmailList()
```

- Using Jython list:

```
AdminTask.getEmailList()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEmailList('-interactive')
```

getSendEmail

The getSendEmail command displays whether or not the audit notification object sends an email if the audit subsystem fails. If the notification monitor is not configured, the audit notification object is not active and the command returns a null value.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system is configured to send an email to the distribution list.

Batch mode example usage

- Using Jython string:

```
AdminTask.getSendEmail()
```

- Using Jython list:

```
AdminTask.getSendEmail()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getSendEmail('-interactive')
```

getAuditNotificationRef

The getAuditNotificationRef command retrieves the reference ID for the active audit notification object. If the notification monitor is not configured, the audit notification object is not active and the command returns a null value.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns the reference ID of the audit notification object if it is active, as the following sample output displays:

```
WSNotification_1184690835390
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditNotificationRef()
```

- Using Jython list:

```
AdminTask.getAuditNotificationRef()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditNotificationRef('-interactive')
```

getAuditNotificationName

The getAuditNotificationName command retrieves the unique name for the active audit notification object. If the notification monitor is not configured, the audit notification object is not active and the command returns a null value.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns the unique name of the audit notification object, as the following sample output displays:

```
mynotification
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditNotificationName()
```

- Using Jython list:

```
AdminTask.getAuditNotificationName()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditNotificationName('-interactive')
```

isSendEmailEnabled

The isSendEmailEnabled command determines if the system is configured to send an email if the security auditing subsystem fails.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if email notification is enabled.

Batch mode example usage

- Using Jython string:

```
AdminTask.isSendEmailEnabled()
```

- Using Jython list:

```
AdminTask.isSendEmailEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isSendEmailEnabled('-interactive')
```

isAuditNotificationEnabled

The isAuditNotificationEnabled command determines whether the security auditing system notifications are enabled.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if security auditing system notifications are enabled.

Batch mode example usage

- Using Jython string:

```
AdminTask.isAuditNotificationEnabled()
```

- Using Jython list:

```
AdminTask.isAuditNotificationEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isAuditNotificationEnabled()
```

listAuditNotifications

The listAuditNotifications command retrieves the attributes for each audit notification object that is configured in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes for each configured audit notification object, as the following sample output displays:

```
{{name mynotification}
{sslConfig {}}
{logToSystemOut true}
{_Websphere_Config_Data_Id cells/CHEYENNENode04Cell|audit.xml#WSNotification_1184690835390}
{emailList sweetshadow@us.ibm.com(smtp-server.us.ibm.com)}
{sendEmail true}
{notificationRef WSNotification_1184690835390}
{_Websphere_Config_Data_Type WSNotification}
{properties {}}
{emailFormat HTML}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditNotifications()
```

- Using Jython list:

```
AdminTask.listAuditNotifications()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditNotifications('-interactive')
```

listAuditNotificationMonitors

The listAuditNotificationMonitors command lists the attributes for the audit notification monitor that is configured in the `audit.xml` file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes for the audit notification monitor, as the following sample output displays:

```
{{name mymonitor}
{enabled true}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditNotificationMonitor_1184695615171}
{_Websphere_Config_Data_Type AuditNotificationMonitor}
{monitorRef AuditNotificationMonitor_1184695615171}
{wsNotification mynotification(cells/Node04Cell|audit.xml#WSNotification_1184690835390)}
{notificationRef WSNotification_1184690835390}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listAuditNotificationMonitors()
```

- Using Jython list:

```
AdminTask.listAuditNotificationMonitors()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAuditNotificationMonitors('-interactive'b)
```

modifyAuditNotification

The modifyAuditNotification command edits the audit notification object in the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-notificationRef
Specifies the reference ID of the audit notification object to edit. (String, required)

Optional parameters

-logToSystemOut
Specifies whether to log notifications to the `SystemOut.log` file. (Boolean, optional)

-sendEmail
Specifies whether to email notifications. (Boolean, optional)

-emailList
Specifies the email address of distribution list where the system sends email notifications. (String, optional)

-emailFormat
Specifies the email format. Specify `HTML` for HTML format or `TEXT` for text format. (String, optional)

Return value

The command returns a value of `true` if the system successfully updates the security auditing system configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditNotification('-notificationRef WSNotification_1184690835390
-logToSystemOut false -sendEmail true -emailList admin@mycompany.com(smtp-server.mycompany.com)
-emailFormat TEXT')
```

- Using Jython list:

```
AdminTask.modifyAuditNotification(['-notificationRef', 'WSNotification_1184690835390',
'-logToSystemOut', 'false', '-sendEmail', 'true', '-emailList',
'admin@mycompany.com(smtp-server.mycompany.com)', '-emailFormat', 'TEXT'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditNotification('-interactive')
```

modifyAuditNotificationMonitor

The modifyAuditNotificationMonitor command edits the audit notification monitor configuration for the security auditing system.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-monitorRef
Specifies the reference ID of the audit notification monitor of interest. (String, required)

Optional parameters

-notificationRef
Specifies the reference ID of the audit notification object. (String, optional)

-enable
Specifies whether to enable the audit notification monitor. (Boolean, optional)

Return value

The command returns a value of `true` if the system successfully updates the audit notification monitor configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditNotificationMonitor('-monitorRef AuditNotificationMonitor_1184695615171
-notificationRef WSNotification_1184690835390 -enable true')
```

- Using Jython list:

```
AdminTask.modifyAuditNotificationMonitor(['-monitorRef', 'AuditNotificationMonitor_1184695615171',
'-notificationRef', 'WSNotification_1184690835390', '-enable', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditNotificationMonitor('-interactive')
```

setEmailList

The setEmailList command specifies the distribution list to send email notifications to if the security auditing subsystem fails.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-emailList
Specifies the email address or email distribution list to send audit notifications to. (String, required)

Return value

The command returns a value of `true` if the system successfully sets the email notification list for the notification object.

Batch mode example usage

- Using Jython string:

```
AdminTask.setEmailList('[-emailList admin@mycompany.com(smtp-server.mycompany.com)]')
```

- Using Jython list:

```
AdminTask.setEmailList(['-emailList', 'admin@mycompany.com(smtp-server.mycompany.com)'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.setEmailList('-interactive')
```

setSendEmail

The setSendEmail command enables or disables email notifications for the security auditing system.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-enable
Specifies whether to enable the system to send audit notifications by email. (Boolean, required)

Return value

The command returns a value of `true` if the system successfully modifies the configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.setSendEmail('-enable true')
```

- Using Jython list:

```
AdminTask.setSendEmail(['-enable', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setSendEmail('-interactive')
```

AuditPolicyCommands command group for the AdminTask object

You can use the Jython scripting language to manage the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditPolicyCommands group to enable and configure the security auditing system.

Use the following commands to configure, query, and manage the security auditing system:

- "disableAudit"
- "disableVerboseAudit" on page 816
- "enableAudit" on page 816
- "enableVerboseAudit" on page 817
- "getAuditPolicy" on page 817
- "getAuditSystemFailureAction" on page 818
- "getAuditorId" on page 819
- "isAuditEnabled" on page 819
- "isVerboseAuditEnabled" on page 820
- "mapAuditGroupIDsOfAuthorizationGroup" on page 820
- "modifyAuditPolicy" on page 820
- "setAuditSystemFailureAction" on page 822
- "resetAuditSystemFailureAction" on page 822
- "setAuditorId" on page 823
- "setAuditorPwd" on page 823

disableAudit

The disableAudit command disables security auditing in the `audit.xml` configuration file.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully disables security auditing.

Batch mode example usage

- Using Jython string:

```
AdminTask.disableAudit()
```

- Using Jython list:

```
AdminTask.disableAudit()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.disableAudit('-interactive')
```

disableVerboseAudit

The disableVerboseAudit command disables the verbose capture of audit data for the security auditing system.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully disables the verbose capture of audit data.

Batch mode example usage

- Using Jython string:

```
AdminTask.disableVerboseAudit()
```

- Using Jython list:

```
AdminTask.disableVerboseAudit()
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.disableVerboseAudit('-interactive')
```

enableAudit

The enableAudit command enables security auditing in the `audit.xml` configuration file. By default, security auditing is disabled.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully enables security auditing.

Batch mode example usage

- Using Jython string:

```
AdminTask.enableAudit()
```

- Using Jython list:

```
AdminTask.enableAudit()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.enableAudit('-interacive')
```

enableVerboseAudit

The enableVerboseAudit command sets the security auditing system to perform verbose capture of audit data.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully sets the security auditing system to perform verbose capture of audit data.

Batch mode example usage

- Using Jython string:

```
AdminTask.enableVerboseAudit()
```

- Using Jython list:

```
AdminTask.enableVerboseAudit()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.enableVerboseAudit('-interactive')
```

getAuditPolicy

The getAuditPolicy command retrieves each attribute that is associated with the audit policy in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a list of attributes for the security auditing system, as the following sample output displays:

```
{{auditEventFactories {{{name auditEventFactoryImpl_1}
{properties {}}
{className com.ibm.ws.security.audit.AuditEventFactoryImpl}
{auditServiceProvider auditServiceProviderImpl_1(cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608)}
{auditSpecifications DefaultAuditSpecification_1(cells/Node04Cell|audit.xml#AuditSpecification_1173199825608)
DefaultAuditSpecification_2(cells/Node04Cell|audit.xml#AuditSpecification_1173199825609)
DefaultAuditSpecification_3(cells/Node04Cell|audit.xml#AuditSpecification_1173199825610)
DefaultAuditSpecification_4(cells/Node04Cell|audit.xml#AuditSpecification_1173199825611)}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditEventFactory_1173199825608}
{_Websphere_Config_Data_Type AuditEventFactory}}}}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditPolicy_1173199825608}
{auditServiceProviders {{{auditSpecifications
DefaultAuditSpecification_1(cells/Node04Cell|audit.xml#AuditSpecification_1173199825608)
DefaultAuditSpecification_2(cells/Node04Cell|audit.xml#AuditSpecification_1173199825609)
DefaultAuditSpecification_3(cells/Node04Cell|audit.xml#AuditSpecification_1173199825610)
DefaultAuditSpecification_4(cells/Node04Cell|audit.xml#AuditSpecification_1173199825611)}
{name auditServiceProviderImpl_1}
{_Websphere_Config_Data_Id cells/Node04Cell|audit.xml#AuditServiceProvider_1173199825608}
```

```
{maxFileSize 1}
{_Websphere_Config_Data_Type AuditServiceProvider}
{fileLocation ${PROFILE_ROOT}/logs/server1}
{className com.ibm.ws.security.audit.BinaryEmitterImpl}
{properties {}}
{eventFormatterClass {}}
{maxLogs 100}}}}
{securityXmlSignerCertAlias auditSignCert}
{properties {}}
{securityXmlSignerScopeName (cell):Node04Cell:(node):Node04}
{auditorPwd SweetShadowsPwd}
{_Websphere_Config_Data_Type AuditPolicy}
{securityXmlSignerKeyStoreName NodeDefaultSignersStore}
{verbose false}
{auditPolicy WARN}
{encrypt false}
{managementScope {}}
{encryptionCert {}}
{batching false}
{auditorId SweetShadow}
{auditEnabled false}
{sign true}}
```

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditPolicy()
```

- Using Jython list:

```
AdminTask.getAuditPolicy()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditPolicy('-interactive')
```

getAuditSystemFailureAction

The getAuditSystemFailureAction command displays the action that the application server takes if a failure occurs in the security auditing subsystem.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a string that describes the action that the application server takes if the security auditing subsystem fails. Possible values are WARN, NOWARN, or FATAL. The following table describes the behavior associated with each action that the application server takes if the security auditing subsystem fails:

WARN	Specifies that the application server should notify the auditor, stop security auditing, and continue to run the application server process.
NOWARN	Specifies that the application server should not notify the auditor, but should stop security auditing and continue to run the application server process
FATAL	Specifies that the application server should notify the auditor, stop security auditing, and stop the application server process.

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditSystemFailureAction()
```

- Using Jython list:

```
AdminTask.getAuditSystemFailureAction()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditSystemFailureAction('-interactive')
```

getAuditorId

The getAuditorId command retrieves the name of the user who is assigned as the auditor.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns the name of the user who is assigned as the auditor.

Batch mode example usage

- Using Jython string:

```
AdminTask.getAuditorId()
```

- Using Jython list:

```
AdminTask.getAuditorId()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getAuditorId('-interactive')
```

isAuditEnabled

The isAuditEnabled command determines whether the security auditing is enabled in your configuration. By default, auditing is not enabled in the `audit.xml` configuration file.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if security auditing is enabled in your environment. If the command returns a value of `false`, security auditing is disabled.

Batch mode example usage

- Using Jython string:

```
AdminTask.isAuditEnabled()
```

- Using Jython list:

```
AdminTask.isAuditEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isAuditEnabled('-interactive')
```

isVerboseAuditEnabled

The isVerboseAuditEnabled command determines whether or not the security auditing system verbosely captures audit data.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the security auditing system is configured to verbosely capture audit data.

Batch mode example usage

- Using Jython string:

```
AdminTask.isVerboseAuditEnabled()
```

- Using Jython list:

```
AdminTask.isVerboseAuditEnabled()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.isVerboseAuditEnabled('-interactive')
```

mapAuditGroupIDsOfAuthorizationGroup

The mapAuditGroupIDsOfAuthorizationGroup command maps the special subjects to users in the registry.

The user must have the monitor administrative role to run this command.

Target object

None.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.mapAuditGroupIDsOfAuthorizationGroup()
```

- Using Jython list:

```
AdminTask.mapAuditGroupIDsOfAuthorizationGroup()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.mapAuditGroupIDsOfAuthorizationGroup('-interactive')
```

modifyAuditPolicy

The modifyAuditPolicy command modifies the audit policy attributes in the `audit.xml` configuration file. You can use this command to modify one or multiple attributes.

The user must have the auditor administrative role to run this command.

Target object

None.

Optional parameters

-auditEnabled
Specifies whether security auditing is enabled in your configuration. (Boolean, optional)

-auditPolicy
Specifies the action that the application server takes if the security auditing subsystem fails. (String, optional)

The following table describes the valid values for this parameter:

WARN	Specifies that the application server should notify the auditor, stop security auditing, and continue to run the application server process.
NOWARN	Specifies that the application server should not notify the auditor, but should stop security auditing and continue to run the application server process
FATAL	Specifies that the application server should notify the auditor, stop security auditing, and stop the application server process.

-auditorId
Specifies the name of the user that the system assigns as the auditor. (String, optional)

-auditorPwd
Specifies the password for the auditor id. (String, optional)

-sign
Specifies whether to sign audit records. Use the AuditSigningCommands command group to configure signing settings. (Boolean, optional)

-encrypt
Specifies whether to encrypt audit records. Use the AuditEncryptionCommands command group to configure encryption settings. (Boolean, optional)

-verbose
Specifies whether to capture verbose audit data. (Boolean, optional)

-encryptionCert
Specifies the reference ID of the certificate to use for encryption. Specify this parameter if you set the -encrypt parameter to `true`. (String, optional)

Return value

The command returns a value of `true` if the system successfully updates the security auditing system policy.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyAuditPolicy('-auditEnabled true -auditPolicy NOWARN -auditorId
testuser -auditorPwd testuserpwd -sign false -encrypt false -verbose false')
```

- Using Jython list:

```
AdminTask.modifyAuditPolicy(['-auditEnabled', 'true', '-auditPolicy', 'NOWARN',
'-auditorId', 'testuser', '-auditorPwd', 'testuserpwd', '-sign', 'false', '-encrypt',
'false', '-verbose', 'false'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyAuditPolicy('-interactive')
```

setAuditSystemFailureAction

The setAuditSystemFailureAction command sets the action that the application server takes if the security auditing subsystem fails.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-action
Specifies the action to take if the security auditing subsystem fails. (String, required)

The following table describes the valid values for this parameter:

WARN	Specifies that the application server should notify the auditor, stop security auditing, and continue to run the application server process.
NOWARN	Specifies that the application server should not notify the auditor, but should stop security auditing and continue to run the application server process
FATAL	Specifies that the application server should notify the auditor, stop security auditing, and stop the application server process.

Return value

The command returns a value of `true` if the system successfully updates the security auditing system policy.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAuditSystemFailureAction('-action NOWARN')
```

- Using Jython list:

```
AdminTask.setAuditSystemFailureAction(['-action', 'NOWARN'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setAuditSystemFailureAction('-interactive')
```

resetAuditSystemFailureAction

The resetAuditSystemFailureAction command sets the action that the application server takes if the security auditing system fails to the `NOWARN` setting.

The user must have the auditor administrative role to run this command.

Target object

None.

Return value

The command returns a value of `true` if the system successfully updates your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.resetAuditSystemFailureAction()
```

- Using Jython list:

```
AdminTask.resetAuditSystemFailureAction()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.resetAuditSystemFailureAction('-interactive')
```

setAuditorId

The setAuditorId command sets the name of the user to assign as the auditor.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-name
Specifies the name of the user to assign as the auditor. (String, required)

Return value

The command returns a value of `true` if the system successfully updates your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTAsk.setAuditorId('-name myAdmin')
```

- Using Jython list:

```
AdminTAsk.setAuditorId(['-name', 'myAdmin'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTAsk.setAuditorId('-interactive')
```

setAuditorPwd

The setAuditorPwd command sets the password for the auditor.

The user must have the auditor administrative role to run this command.

Target object

None.

Required parameters

-password
Specifies the password for the user assigned as the auditor. (String, required)

Return value

The command returns a value of `true` if the system successfully updates your configuration.

Batch mode example usage

- Using Jython string:

```
AdminTask.setAuditorPwd('-password myAdminPassword')
```

- Using Jython list:

```
AdminTask.setAuditorPwd(['-password', 'myAdminPassword'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setAuditorPwd('-interactive')
```

AuditEventFormatterCommands command group for the AdminTask object

You can use the Jython scripting language to manage the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditEventFormatterCommands group to manage the event formatter for the audit service provider.

Use the following commands to display information about the audit event formatter:

- "getEventFormatterClass"

getEventFormatterClass

The getEventFormatterClass command retrieves the class name of the event formatter specified by the reference ID of the binary file audit service provider of interest.

The user must have the monitor administrative role to run this command.

Target object

None.

Required parameters

-emitterRef
Specifies the reference ID of the audit service provider implementation of interest. (String, required)

Return value

The command returns a null value if the event formatter class is not defined for the audit service provider implementation of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getEventFormatterClass('-emitterRef AuditServiceProvider_1173199825608')
```

- Using Jython list:

```
AdminTask.getEventFormatterClass(['-emitterRef', 'AuditServiceProvider_1173199825608'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getEventFormatterClass('-interactive')
```

AuditReaderCommands command group for the AdminTask object

You can use the Jython scripting language to manage the security auditing system with the wsadmin tool. Use the commands and parameters in the AuditReaderCommands group to display audit record information from the binary audit log.

Use the following commands to query the binary audit log:

- "binaryAuditLogReader"
- "showAuditLogEncryptionInfo" on page 826

binaryAuditLogReader

The binaryAuditLogReader command reads the default binary audit log and generates an HTML report based on the parameters you provide. You must use the auditor security role to use this command.

Target object

None.

Required parameters

-fileName
Specifies the fully qualified file name for the binary audit log. (String, required)

-outputLocation
Specifies the location of the HTML report that the command generates. (String, required)

Optional parameters

-reportMode
Specifies the type of report to generate. Valid values include `basic`, `complete`, or `custom`. The `basic` report provides the following configuration information:

- creationTime
- action
- progName
- registryType
- domain
- realm
- remoteAddr
- remotePort
- remoteHost
- resourceName
- resourceType
- resourceUniqueId

The `complete` report provides the data included by the default report type and each additional datapoint of interest. The `custom` report allows you to specify only the datapoints you choose to see generated in the report. The default value is basic. (String, optional)

-eventFilter
Specifies the audit types to read and report. Specify one or more audit event types. If you specify more than one value for the eventFilter parameter, separate each audit event type with a colon character (`:`). (String, optional)

-outcomeFilter
Specifies the audit event outcomes to read and report. Specify one or more audit event outcomes. If you specify more than one value for the outcomeFilter parameter, separate each audit event outcome with a colon character (`:`). (String, optional)

-sequenceFilter
Specifies a list of beginning and ending sequence numbers. Use the `a:b` syntax, where `a`, the starting sequence number where the HTML report begins, and is less than or equal to `b`, the sequence number where the HTML report ends. A single sequence may also be specified, such as `-sequenceFilter 10`, to only generate a report for the tenth record. (String, optional)

-timeStampFilter
Specifies the time stamp range of records to read and report. Use the `a:b` syntax, where `a` and `b` are strings in the format `java.text.SimpleDateFormat("MMddhhmmyyyy")`. You can also specify a single timestamp. (String, optional)

-keyStorePassword
Specifies password to open the keystore. (String, optional)

-dataPoints
Specifies a list of specific audit data to use to generate the report. Use this option only when you set the reportMode parameter as `custom`. If you specify multiple data points, separate each data point with a colon character (`:`). (String, optional)

Return value

The command returns the HTML report based on the values specified for each parameter to the location specified by the outputLocation parameter.

Batch mode example usage

- Using Jython string:

```
AdminTask.binaryAuditLogReader('[-fileName myFileName -reportMode basic
-keyStorePassword password123 -outputLocation /binaryLogs]')
```

- Using Jython list:

```
AdminTask.binaryAuditLogReader(['-fileName', 'myFileName', '-reportMode', 'basic',
'-keyStorePassword', 'password123', '-outputLocation', '/binaryLogs'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.binaryAuditLogReader('-interactive')
```

showAuditLogEncryptionInfo

The showAuditLogEncryptionInfo command displays information about the keystore that the auditing system uses to encrypt audit records. Use this information as a hint of the keystore password in order to decrypt encrypted audit logs in the binary audit log.

Target object

None.

Required parameters

-fileName
Specifies the fully qualified path of the binary audit log. (String, required)

Return value

The command returns the certificate alias and the fully qualified path to the keystore of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.showAuditLogEncryptionInfo('-fileName myFileName')
```

- Using Jython list:

```
AdminTask.showAuditLogEncryptionInfo(['-fileName', 'myFileName'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.showAuditLogEncryptionInfo('-interactive')
```

SSLMigrationCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to migrate key store configurations. Use the commands in the SSLMigrationCommands group to convert self-signed certificates to chained personal certificates and to enable writable key rings.

The SSLMigrationCommands command group for the AdminTask object includes the following commands:

- "convertSelfSignedCertificatesToChained command"
- "enableWritableKeyrings command" on page 828
- "convertSSLConfig command" on page 829

convertSelfSignedCertificatesToChained command

The convertSelfSignedCertificatesToChained command converts specific self-signed certificates to chained personal certificates.

Note: Chained certificates are the default certificate type in Websphere Application Server Version 7.0. The convertSelfSignedCertificatesToChained command takes information from the self-signed certificate—such as issued-to DN, size, and life span—and creates a chained certificate with the same information. The new chained certificate replaces the self-signed certificate. Signer certificates from the self-signed certificate that are distributed across the security configuration are replaced with the signer certificates from the root certificate used to sign the chained certificate.

Syntax

The command has the following syntax:

```
wsadmin>$AdminTask convertSelfSignedCertificatesToChained
                    [-certificateReplacementOption ALL_CERTIFICATES | DEFAULT_CERTIFICATES | KEYSTORE_CERTIFICATES]
                    [-keyStoreName keystore_name]
                    [-keyStoreScope keystore_scope]
                    [-rootCertificateAlias alias_name]
```

Required parameters

certificateReplacementOption

Specifies the convert self-signed certificates replacement options. (String, required)

Specify the value for the parameter as one of the following options:

ALL_CERTIFICATES

This option looks for all self-signed certificates in all keystores with in the specified scope.

The scope can be provided in the -keyStoreScope parameter. If no scope is provided using the -keyStoreScope parameter, all scopes are visited.

DEFAULT_CERTIFICATES

This option looks for self-signed certificates in the default CellDefaultKeyStore and NodeDefaultKeyStore keystores within the specified scope.

The scope can be provided with the -keyStoreScope parameter. If no scope is provided using the -keyStoreScope parameter, all scopes are visited.

KEYSTORE_CERTIFICATES

This option replaces only those self-signed certificates in the keystore that are specified by the -keyStoreName parameter.

If no scope is provided using the -keyStoreScope parameter, the default scope is used.

Optional parameters

keyStoreName
Specifies the name of a keystore in which to look for self-signed certificates to convert. Use this parameter with the KEYSTORE_CERTIFICATES option on the certificateReplacementOption parameter. (String, optional)

keyStoreScope
Specifies the name of the scope in which to look for the self-signed certificates to convert. (String, optional)

rootCertificateAlias
Specifies the root certificate to use from the default root store used to sign the chained certificate. The default value is `root`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask convertSelfSignedCertificatesToChained {-certificateReplacementOption ALL_CERTIFICATES -keyStoreName testKS}
```

- Using Jython string:

```
AdminTask.convertSelfSignedCertificatesToChained('[-certificateReplacementOption ALL_CERTIFICATES -keyStoreName testKS]')
```

- Using Jython list:

```
AdminTask.convertSelfSignedCertificatesToChained(['-certificateReplacementOption', 'ALL_CERTIFICATES', '-keyStoreName', 'testKS'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exchangeSigners {-interactive}
```

- Using Jython:

```
AdminTask.exchangeSigners('-interactive')
```

enableWritableKeyrings command

The enableWritableKeyrings command modifies the keystore and enables writable SAF support. The system uses this command during migration. The command creates additional writable keystore objects for the control region and servant region key rings for SSL keystores.

Required parameters

-keyStoreName
Specifies the name that uniquely identifies the keystore that you want to delete. (String, required)

Optional parameters

-controlRegionUser
Specifies the control region user to use to enable writable key rings. (String, optional)

-servantRegionUser
Specifies the servant region user to enable writable key rings. (String, optional)

-scopeName
Specifies the name that uniquely identifies the management scope, for example: `(cell):localhostNode01Cell`. (String, optional)

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.enableWritableKeyrings('[-keyStoreName testKS -controlRegionUser CRUser1 -servantRegionUser SRUser1]')
```

- Using Jython list:

```
AdminTask.enableWritableKeyrings(['-keyStoreName', 'testKS', '-controlRegionUser', 'CRUser1', '-servantRegionUser', 'SRUser1'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.enableWritablekeyrings('-interactive')
```

convertSSLConfig command

The convertSSLConfig command migrates existing SSL configurations to the new configuration object format for SSL configurations.

Required parameters

-sslConversionOption
Specifies how the system converts the SSL configuration. Specify the `CONVERT_SSLCONFIGS` value to convert the SSL configuration objects from the previous SSL configuration object to the new SSL configuration object. Specify the `CONVERT_TO_DEFAULT` value to convert the SSL configuration to a centralized SSL configuration, which also removes the SSL configuration direct referencing from the servers.

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.convertSSLConfig('[-keyStoreName testKS -controlRegionUser CRUser1 -servantRegionUser SRUser1]')
```

- Using Jython list:

```
AdminTask.convertSSLConfig(['-keyStoreName', 'testKS', '-controlRegionUser', 'CRUser1', '-servantRegionUser', 'SRUser1'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.convertSSLConfig('-interactive')
```

Related concepts

Key management for cryptographic uses
WebSphere Application Server provides a framework for managing keys (secret keys or key pairs) that applications use to perform cryptographic operations on data. The key management framework provides an application programming interface (API) for retrieving these keys. Keys are managed in keystores so the keystore type can be supported by WebSphere Application Server, provided that the keystores can store the referenced key type. You can configure keys and scope keystores so that they are visible only to particular processes, nodes, clusters, and so on.

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

"Automating SSL configurations using scripting" on page 633
SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

"Creating an SSL configuration at the node scope using scripting" on page 631
An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

Related reference

"KeyStoreCommands command group for the AdminTask object" on page 654
You can use the Jython or Jacl scripting languages to configure keystores with the wsadmin tool. A keystore is created by the application server during install and can contain cryptographic keys or certificates. The commands and parameters in the KeyStoreCommands group can be used to create, delete, and manage keystores.

IdMgrConfig command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure the virtual member manager with the wsadmin tool. The commands and parameters in the IdMgrConfig group can be used to create and manage your entity type configuration.

The IdMgrConfig command group for the AdminTask object includes the following commands:

- "deleteIdMgrSupportedEntityType"
- "getIdMgrSupportedEntityType" on page 831
- "listIdMgrSupportedEntityTypes" on page 832
- "resetIdMgrConfig" on page 832
- "showIdMgrConfig" on page 833
- "updateIdMgrLDAPBindInfo" on page 833
- "updateIdMgrSupportedEntityType" on page 834

deleteIdMgrSupportedEntityType

The **deleteIdMgr Supported EntityType** command deletes the supported entity type configuration that you specify.

Parameters and return values

-name
The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrSupported EntityType {-name entity1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrSupported EntityType ('[-name entity1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgr SupportedEntityType ([' -name', 'entity1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgr SupportedEntityType {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgr SupportedEntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgr SupportedEntityType (['-interactive'])
```

getIdMgrSupportedEntityType

The **getIdMgr Supported EntityType** command returns the configuration of the supported entity type that you specify.

Parameters and return values

-name
The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrSupported EntityType {-name entity1}
```

- Using Jython string:

```
AdminTask.getIdMgrSupported EntityType ('[-name entity1]')
```

- Using Jython list:

```
AdminTask.getIdMgrSupported EntityType (['-name', 'entity1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrSupported EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrSupported EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrSupported EntityType (['-interactive'])
```

listIdMgrSupportedEntityTypes

The **listIdMgr Supported EntityTypes** command lists all of the supported entity types that are configured.

Parameters and return values
- Parameters: None
- Returns: A list that contains the names of the supported entity types.

Examples

Batch mode example usage:
- Using Jacl:
```
$AdminTask listIdMgr SupportedEntityTypes
```
- Using Jython string:
```
AdminTask.listIdMgr SupportedEntityTypes()
```
- Using Jython list:
```
AdminTask.listIdMgr SupportedEntityTypes()
```

Interactive mode example usage:
- Using Jacl:
```
$AdminTask listIdMgrSupported EntityTypes {-interactive}
```
- Using Jython string:
```
AdminTask.listIdMgrSupported EntityTypes ('[-interactive]')
```
- Using Jython list:
```
AdminTask.listIdMgrSupported EntityTypes (['-interactive'])
```

resetIdMgrConfig

The **resetId MgrConfig** command resets the current configuration to the last configuration that was saved.

Parameters and return values
- Parameters: None
- Returns: None

Examples

Batch mode example usage:
- Using Jacl:
```
$AdminTask resetIdMgrConfig
```
- Using Jython string:
```
AdminTask.resetIdMgrConfig()
```
- Using Jython list:
```
AdminTask.resetIdMgrConfig()
```

Interactive mode example usage:
- Using Jacl:
```
$AdminTask resetIdMgrConfig {-interactive}
```
- Using Jython string:
```
AdminTask.resetIdMgrConfig ('[-interactive]')
```
- Using Jython list:

```
AdminTask.resetIdMgrConfig (['-interactive'])
```

showIdMgrConfig

The **showId MgrConfig** command returns the current configuration XML in string format.

Parameters and return values

- Parameters: None
- Returns: None

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask showIdMgrConfig
  ```
- Using Jython string:
  ```
  AdminTask.showIdMgrConfig()
  ```
- Using Jython list:
  ```
  AdminTask.showIdMgrConfig()
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask showIdMgrConfig {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.showIdMgrConfig ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.showIdMgrConfig (['-interactive'])
  ```

updateIdMgrLDAPBindInfo

The **updateIdMgrLDAPBindInfo** command updates the LDAP password under the federated repository. The change is also reflected in the wimconfig.xml file.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jython:
  ```
  AdminTask.updateIdMgrLDAPBindInfo('[-id id1 -bindDN cn=root -bindPassword myPassword22]')
  ```
- Using Jython list:
  ```
  AdminTask.updateIdMgrLDAPBindInfo(['-id id1 -bindDN cn=root -bindPassword myPassword22'])
  ```
- Using Jacl:
  ```
  $AdminTask updateIdMgrLDAPBindInfo {-id id1 -bindDN cn=root -bindPassword myPassword22}
  ```

Interactive mode example usage:

- Using Jython:
  ```
  AdminTask.updateIdMgrLDAPBindInfo(['-interactive'])
  ```
- Using Jacl:

```
$AdminTask updateIdMgrLDAPBindInfo {-interactive}
```

updateIdMgrSupportedEntityType

The **updateIdMgr Supported EntityType** command updates the configuration that you specify for a supported entity type.

Parameters and return values

-name
The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required)

-defaultParent
The default parent node for the supported entity type. (String, optional)

-rdnProperties
The RDN™ attribute name for the supported entity type in the entity domain name. To reset all the values of the rdnProperties parameter, specify a blank string (""). (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrSupported EntityType {-name entity1}
```

- Using Jython string:

```
AdminTask.updateIdMgrSupported EntityType ('[-name entity1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrSupported EntityType (['-name', 'entity1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrSupported EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrSupported EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrSupported EntityType (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

"IdMgrRepositoryConfig command group for the AdminTask object"
You can use the Jython or Jacl scripting languages to configure security. The commands and parameters in the IdMgrRepositoryConfig group can be used to create and manage the virtual member manager and LDAP directory properties.

"IdMgrRealmConfig command group for the AdminTask object" on page 882
You can use the Jython or Jacl scripting languages to configure the member manager realm and realms. The commands and parameters in the IdMgrRealmConfig group can be used to create and manage your realm configuration.

IdMgrRepositoryConfig command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security. The commands and parameters in the IdMgrRepositoryConfig group can be used to create and manage the virtual member manager and LDAP directory properties.

The IdMgrRepositoryConfig command group for the AdminTask object includes the following commands:

- "addIdMgrLDAPAttr" on page 836
- "addIdMgrLDAPBackupServer" on page 837
- "addIdMgrLDAPEntityType" on page 838
- "addIdMgrLDAPEntityTypeRDNAttr" on page 839
- "addIdMgrLDAPGroupDynamicMemberAttr" on page 839
- "addIdMgrLDAPGroupMemberAttr" on page 840
- "addIdMgrLDAPServer" on page 841
- "addIdMgrRepositoryBaseEntry" on page 842
- "createIdMgrCustomRepository" on page 843
- "createIdMgrDBRepository" on page 843
- "createIdMgrFileRepository" on page 845
- "createIdMgrLDAPRepository" on page 846
- "deleteIdMgrLDAPAttr" on page 847
- "deleteIdMgrLDAPEntityType" on page 848
- "deleteIdMgrLDAPEntityTypeRDNAttr" on page 848
- "deleteIdMgrLDAPGroupConfig" on page 849
- "deleteIdMgrLDAPGroupMemberAttr" on page 849
- "deleteIdMgrLDAPGroupDynamicMemberAttr" on page 850
- "deleteIdMgrLDAPServer" on page 850
- "deleteIdMgrRepository" on page 851
- "deleteIdMgrRepositoryBaseEntry" on page 851
- "getIdMgrLDAPAttrCache" on page 852
- "getIdMgrLDAPContextPool" on page 852
- "getIdMgrLDAPEntityType" on page 853
- "getIdMgrLDAPEntityTypeRDNAttr" on page 853

addIdMgrLDAPAttr

The addIdMgrLDAPAttr command adds an LDAP attribute configuration to the LDAP repository configuration.

Required parameters

-id Specifies the unique ID of the repository. (String, required)

-name
Specifies the name of the LDAP attribute used in the repository LDAP adapter. (String, required)

Optional parameters

-entityTypes
Specifies the entity type which applies the attribute mapping. (String, optional)

-syntax
Specifies the syntax of the LDAP attribute. The default value is `string`. For example, the syntax of the `unicodePwd` LDAP attribute is `octetString`. (String, optional)

-defaultValue
Specifies the default value of the LDAP attribute. If you do not specify this LDAP attribute when you create an entity which this LDAP attribute applies to, the system adds the attribute using this default value. (String, optional)

-defaultAttr
The default attribute of the LDAP attribute. If you do not specify this LDAP attribute when you create an entity which this LDAP attribute applies to, the system uses this value of the default attribute. (String, optional)

-propertyName
Specifies the name of the corresponding federated repository property. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPAttr {-id id1 -name unicodePwd -syntax octetString}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPAttr ('[-id id1 -name unicodePwd -syntax octetString]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPAttr (['-id', 'id1', '-name', 'unicodePwd', '-syntax', 'octetString'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPAttr {-interactive}
```

- Using Jython:

```
AdminTask.addIdMgrLDAPAttr('-interactive')
```

addIdMgrLDAPBackupServer

The **addIdMgrLDAPBackupServer** command sets a backup LDAP server in your configuration.

Required parameters

-id Specifies the unique ID of the repository. (String, required)

-primary_host
Specifies the primary host of the LDAP server. (String, required)

-host
Specifies the host name for the LDAP server. (String, required)

Optional parameters

-port
Specifies the port number for the LDAP server. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPBackupServer {-id id1 -primary_host hostName -host hostName2 -port 4020}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPBackupServer ('[-id id1 -primary_host hostName -host hostName2 -port 4020]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPBackupServer (['-id', 'id1', '-primary_host', 'hostName', '-host', 'hostName2', '-port', '4020'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPBackupServer {-interactive}
```

- Using Jython:

```
AdminTask.addIdMgrLDAPBackupServer('-interactive')
```

addIdMgrLDAPEntityType

The **addIdMgrLDAPEntityType** command adds an LDAP entity type definition.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the entity type. (String, required)

-searchFilter
The search filter that you want to use to search the entity type. (String, optional)

-objectClasses
One or more object classes for the entity type. (String, required)

-objectClassesForCreate
The object class to use when an entity type is created. If the value of this parameter is the same as the objectClass parameter, you do not need to specify this parameter. (String, optional)

-searchBases
The search base or bases to use while searching the entity type. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPEntity Type {-id id1 -name name1 -objectClasses objectclass}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPEntity Type ('[-id id1 -name name1 -objectClasses objectclass]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1', '-obje ctClasses', 'objectclass'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAP EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAP EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAP EntityType (['-interactive'])
```

addIdMgrLDAPEntityTypeRDNAttr

The **addId MgrLDAP EntityType RDNAttr** command adds RDN attribute configuration to an LDAP entity type definition.

Parameters and return values

-id The ID of the repository. (String, required)

-entityTypeName
The name of the entity type. (String, required)

-name
The attribute name that is used to build the relative distinguished name (RDN) for the entity type. (String, required)

-objectClass
The object class to use for the entity type for the relative distinguished name (RDN) attribute name that you specify. Use this parameter to map one entity type to multiple structural object classes. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPEntity TypeRDNAttr {-id id1 -entityTypeName entitytype -name name1}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPEntity TypeRDNAttr ('[-id id1 -entityTypeName entitytype -name name1]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-entityTypeName', 'entity type', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPEntity TypeRDNAttr {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPEntity TypeRDNAttr (['-interactive'])
```

addIdMgrLDAPGroupDynamicMemberAttr

The **addIdMgr LDAPGroup Dynamic Member Attr** command adds a dynamic member attribute configuration to an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required)

-objectClass
The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUniqueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)

-scope
The scope of the member attribute. The valid values for this parameter include the following:

- direct - The member attribute only contains direct members, therefore, this value refers to the member directly contained by the group and not contained through the nested group. For example, if Group1 contains Group2 and Group2 contains User1, then Group2 is a direct member of Group1 but User1 is not a direct member of Group1. Both `member` and `uniqueMember` are direct member attributes.
- nested - The member attribute that contains the direct members and the nested members.

-dummyMember
Indicates that if you create a group without specifying a member, a dummy member will be filled in to avoid creating an exception about missing a mandatory attribute. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPGroup DynamicMemberAttr {-id id1 -name name1 -objectClass objectclass}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPGroup DynamicMemberAttr ('[-id id1 -name name1 -objectClass objectclass]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPGroup DynamicMemberAttr (['-id', 'id1', '-name', 'name1', '-objectClass', 'objectclass'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPGroup DynamicMemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])
```

addIdMgrLDAPGroupMemberAttr

The **addIdMgr LDAPGroup MemberAttr** command adds a member attribute configuration to an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required)

-objectClass
The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUnqiueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)

-scope
The scope of the member attribute. The valid values for this parameter include the following:

- direct - The member attribute only contains direct members, therefore, this value refers to the member directly contained by the group and not contained through the nested group. For example, if Group1 contains Group2 and Group2 contains User1, then Group2 is a direct member of Group1 but User1 is not a direct member of Group1. Both `member` and `uniqueMember` are direct member attributes.

- nested - The member attribute that contains the direct members and the nested members.

-dummyMember
Indicates that if you create a group without specifying a member, a dummy member will be filled in to avoid creating an exception about missing a mandatory attribute. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPGroup MemberAttr {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPGroup MemberAttr ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPGroup MemberAttr (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPGroup MemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPGroup MemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPGroup MemberAttr (['-interactive'])
```

addIdMgrLDAPServer

The **addId MgrLDAP Server** command adds an LDAP server to the LDAP repository ID that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

-host
The host name for the primary LDAP server. (String, required)

-port
The port number for the LDAP server. (Integer, optional)

-bindDN
The binding distinguished name for the LDAP server. (String, optional)

-bindPassword
The binding password. (String, optional)

-authentication
Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional)

-referal
The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional)

-derefAliases
Controls how aliases are dereferenced. The default value is `always`. Valid values include:

- `never` - never deference aliases
- `finding` - deferences aliases only during name resolution
- `searching` - deferences aliases only after name resolution

(String, optional)

-sslEnabled
Indicates to enable SSL or not. The default value is `false`. (Boolean, optional)

-connectionPool
The connection pool. The default value is `false`. (Boolean, optional)

-connectTimeout
The connection timeout in seconds. The default value is `0`. (Integer, optional)

-ldapServerType
The type of LDAP server being used. The default value is `IDS51`. (String, optional)

-sslConfiguration
The SSL configuration. (String, optional)

-certificateMapMode
Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `EXACT_DN`. To use the certificate filter for the mapping, specify `CERTIFICATE_FILTER`. (String, optional)

-certificateFilter
If `certificateMapMode` has the value `CERTIFICATE_FILTER`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAPServer {-id id1 -host  myhost.ibm.com}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAPServer ('[-id id1 -host  myhost.ibm.com]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrLDAP Server {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrLDAP Server ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrLDAP Server (['-interactive'])
```

addIdMgrRepositoryBaseEntry

The **addIdMgr Repository BaseEntry**command adds a base entry to the specified repository.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The distinguished name of a base entry. (String, required)

-nameInRepository
The distinguished name in the repository that uniquely identifies the base entry name. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrRepositoryBaseEntry {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.addIdMgrRepositoryBaseEntry ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.addIdMgrRepositoryBaseEntry (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrRepositoryBaseEntry {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrRepositoryBaseEntry ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addIdMgrRepositoryBaseEntry (['-interactive'])
```

createIdMgrCustomRepository

The **createIdMgrCustomRepository** command creates a custom repository configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-adapterClassName
The implementation class name for the repository adapter. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrCustomRepository {-id id1 -adapterClassName adapterClassName}
```

- Using Jython string:

```
AdminTask.createIdMgrCustomRepository('-id id1 -adapterClassName adapterClassName')
```

- Using Jython list:

```
AdminTask.createIdMgrCustomRepository(['-id', 'id1', '-adapterClassName', 'adapterClassName'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrCustomRepository {-interactive}
```

- Using Jython:

```
AdminTask.createIdMgrCustomRepository('-interactive')
```

createIdMgrDBRepository

The **createIdMgrDBRepository** command creates a database repository configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-dataSourceName
The name of the data source. The default value is `jdbc/wimDS`. (String, required)

-databaseType
The type of the database. The default value is `DB2`. (String, required)

-dbURL
The URL of the database. (String, required)

-dbAdminId
The database administrator ID. (String, required if database type is not Apache Derby.)

-dbAdminPassword
The database administrator password. (String, required if database type is not Apache Derby.)

-adapterClassName
The default value is `com.ibm.ws.wim.adapter.db.DBAdapter`. (String, optional)

-JDBCDriverClass
The JDBC driver class name. (String, optional)

-supportSorting
Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional)

-supportTransactions
Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional)

-isExtIdUnique
Specifies if the external ID is unique. The default value is `true`. (Boolean, optional)

-supportExternalName
Indicates if external names are supported or not. The default value is `false`. (Boolean, optional)

-entityRetrievalLimit
Indicates the value of the retrieval limit on database entries. The default value is `200`. (Integer, optional)

-saltLength
The salt length in bits. The default value is `12`. (Integer, optional)

-encryptionKey
The default value is `rZ15ws0ely9yHk3zCs3sTMv/ho8fY17s`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrDB Repository {-id id1 -dataSourceName datasource name -databaseType DB2}
```

- Using Jython string:

```
AdminTask.createIdMgrDB Repository ('[-id id1 -dataSourceName data sourcename -databaseType DB2]')
```

- Using Jython list:

```
AdminTask.createIdMgrDB Repository (['-id', 'id1', '-dataSourceName', 'data sourcename', '-databaseType', 'DB2'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrDB Repository {-interactive}
```

- Using Jython string:

```
AdminTask.createIdMgrDB Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createIdMgrDB Repository (['-interactive'])
```

createIdMgrFileRepository

The **createId MgrFile Repository** command creates a file repository configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-messageDigest Algorithm
The message digest algorithm that will be used for hashing the password. The default value is `SHA-1`. Valid values include the following: `SHA-1`, `SHA-384`, or `SHA-512`.(String, required)

-adapterClassName
The default value is `com.ibm.ws. wim.adapter. file.was. FileAdapter`. (String, optional)

-supportPaging
Indicates if paging is supported or not. The default value is `false`. (Boolean, optional)

-supportSorting
Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional)

-supportTransactions
Indicates if transaction is supported or not. The default value is `false`. (Boolean, optional)

-isExtIdUnique
Specifies if the external ID is unique or not. The default value is `true`. (Boolean, optional)

-supportExternalName
Indicates if external names are supported or not. The default value is `false`. (Boolean, optional)

-baseDirectory
The base directory where the fill will be created in order to store the data. The default is to be dynamically built during run time using user.install.root and cell name. (String, optional)

-fileName
The file name of the repository. The default value is `fileRegistry.xml`. (String, optional)

-saltLength
The salt length of the randomly generated salt for password hashing. The default value is `12`. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrFile Repository {-id id1 -messageDigestAlgorithm SHA-1}
```

- Using Jython string:

```
AdminTask.createIdMgrFile Repository ('[-id id1 -messageDigestAlgorithm SHA-1]')
```

- Using Jython list:

```
AdminTask.createIdMgrFile Repository (['-id', 'id1', '-messageDigestAlgorithm', 'SHA-1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrFile Repository {-interactive}
```

- Using Jython string:

```
AdminTask.createIdMgrFile Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createIdMgrFile Repository (['-interactive'])
```

createIdMgrLDAPRepository

The **create IdMgrLDAP Repository** command creates an LDAP repository configuration.

Parameters and return values

-id The unique identifier for the repository. (String, required)

-ldapServerType
The type of LDAP server that is being used. The default value is `IDS51`. (String, required)

-adapterClassName
The default value is `com.ibm.ws. wim.adapter. db.DBAda pter`. (String, optional)

-supportSorting
Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional)

-supportPaging
Indicates if paging is supported or not. The default value is `false`. (Boolean, optional)

-supportTransactions
Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional)

-isExtIdUnique
Specifies if the external ID is unique. The default value is `true`. (Boolean, optional)

-supportExternalName
Indicates if external names are supported or not. The default value is `false`. (Boolean, optional)

-authentication
Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional)

-referal
The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional)

-sslEnabled
Indicates to enable SSL or not. The default value is `false`. (Boolean, optional)

-sslConfiguration
The SSL configuration. (String, optional)

-connectionPool
The connection pool. The default value is `false`. (Boolean, optional)

-translateRDN
Indicates to translate RDN or not. The default value is `false`. (Boolean, optional)

-searchTimeLimit
The value of search time limit. (Integer, optional)

-searchCountLimit
The value of search count limit. (Integer, optional)

-searchPageSize
The value of search page size. (Integer, optional)

-returnToPrimaryServer
(Integer, optional)

-primaryServerQueryTimeInterval
(Integer, optional)

-default
If you set this parameter to `true`, the default values will be set for the remaining configuration properties of the LDAP repository. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrLDAP Repository {-id id1 -ldapServerType IDS51}
```

- Using Jython string:

```
AdminTask.createIdMgrLDAP Repository ('[-id id1 -ldapServerType IDS51]')
```

- Using Jython list:

```
AdminTask.createIdMgrLDAP Repository (['-id', 'id1', '-ldapServerType', 'IDS51'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrLDAP Repository {-interactive}
```

- Using Jython string:

```
AdminTask.createIdMgrLDAP Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createIdMgrLDAP Repository (['-interactive'])
```

deleteIdMgrLDAPAttr

The deleteIdMgrLDAPAttr command deletes LDAP attribute configuration data for a specific entity type from the LDAP repository of interest.

Required parameters

-id Specifies the unique ID of the repository. (String, required)

-name
Specifies the name of the LDAP attribute used in the repository LDAP adapter. (String, required)

Optional parameters

-entityTypes
Specifies the entity type which applies the attribute mapping. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAPAttr {-id id1 -name unicodePwd -syntax octetString}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAPAttr ('[-id id1 -name unicodePwd -syntax octetString]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAPAttr (['-id', 'id1', '-name', 'unicodePwd', '-syntax', 'octetString'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAPAttr {-interactive}
```

- Using Jython:

```
AdminTask.deleteIdMgrLDAPAttr('-interactive')
```

deleteIdMgrLDAPEntityType

The **deleteId MgrLDAP EntityType** command deletes the LDAP entity type configuration data for a specified entity type for a specific LDAP repository.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the entity type. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP EntityType {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP EntityType ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP EntityType (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP EntityType (['-interactive'])
```

deleteIdMgrLDAPEntityTypeRDNAttr

The **deleteId MgrLDAP EntityType RDNAttr** command deletes the relative distinguished name (RDN) attribute configuration from an LDAP entity type configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-entityTypeName
The name of the entity type. (String, required)

-name
The attribute name that is used to build the relative distinguished name (RDN) for the entity type. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAPEntity TypeRDNAttr {-id id1 -name name1 -entityTypeName entityType}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP EntityTypeRDNAttr ('[-id id1 -name name1 -entityType Name entityType]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-name', 'name1', '-entity TypeName', 'entityType'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAPEntity TypeRDNAttr {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr (['-interactive'])
```

deleteIdMgrLDAPGroupConfig

The **deleteId MgrLDAP GroupConfig** command deletes the LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP GroupConfig {-id id1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP GroupConfig ('[-id id1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP GroupConfig (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP GroupConfig {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP GroupConfig ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP GroupConfig (['-interactive'])
```

deleteIdMgrLDAPGroupMemberAttr

The **deleteIdMgr LDAPGroup MemberAttr** command deletes a member attribute configuration from an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP GroupMemberAttr {-id id1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP GroupMemberAttr ('[-id id1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP GroupMemberAttr (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP GroupMemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP GroupMemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP GroupMemberAttr (['-interactive'])
```

deleteIdMgrLDAPGroupDynamicMemberAttr

The **deleteIdMgr LDAPGroup Dynamic MemberAttr** command deletes a dynamic member attribute configuration from an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the LDAP attribute that is used as the group member attribute. For example, `memberURL`. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP GroupDynamicMemberAttr {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP GroupDynamicMemberAttr ('[ -id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP GroupDynamicMemberAttr ([' -id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAPGroup DynamicMemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])
```

deleteIdMgrLDAPServer

The **deleteId MgrLDAP Server** command deletes the configuration for the LDAP server that you specify from the LDAP repository ID that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

-host
The host name for the primary LDAP server. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP Server {-id id1 -host myhost.ibm.com}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP Server ('[-id id1 -host myhost.ibm.com]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP Server (['-id', 'id1', '-host', 'myhost.ibm.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrLDAP Server {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrLDAP Server ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrLDAP Server (['-interactive'])
```

deleteIdMgrRepository

The **deleteIdMgr Repository** command deletes a repository that you specify.

Parameters and return values

-id The ID of the repository. Valid values include existing repository IDs. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgr Repository {-id id1}
```

- Using Jython string:

```
AdminTask.deleteIdMgr Repository ('[-id id1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgr Repository (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrRepos itory {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrReposi tory ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrReposi tory (['-interactive'])
```

deleteIdMgrRepositoryBaseEntry

The **deleteIdMgr Repository BaseEntry** command deletes a base entry from the specified repository.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The distinguished name of a base entry. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrRepository BaseEntry {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.deleteIdMgrRepository BaseEntry ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrRepository BaseEntry (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteIdMgrRepository BaseEntry {-interactive}
```

- Using Jython string:

```
AdminTask.deleteIdMgrRepository BaseEntry ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteIdMgrRepository BaseEntry (['-interactive'])
```

getIdMgrLDAPAttrCache

The **getIdMgr LDAPAttr Cache** command returns the LDAP attribute cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPAttr Cache {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPAttr Cache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPAttr Cache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAP AttrCache {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPAttr Cache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPAttr Cache (['-interactive'])
```

getIdMgrLDAPContextPool

The **getIdMgr LDAP Context Pool** command returns the LDAP context pool configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPContext Pool {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPContext Pool ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPContext Pool (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPCon textPool {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPCon textPool ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPCon textPool (['-interactive'])
```

getIdMgrLDAPEntityType

The **getIdMgr LDAP EntityType** command returns the LDAP entity type configuration data.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the entity type. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPEntity Type {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPEntity Type ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPEn tityType {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPEn tityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPEn tityType (['-interactive'])
```

getIdMgrLDAPEntityTypeRDNAttr

The **getIdMgr LDAPEntity TypeRDNAttr** command returns the relative distinguished name (RDN) attribute configuration for an LDAP entity type definition.

Parameters and return values

-id The ID of the repository. (String, required)

-entityTypeName
The name of the entity name. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPEntity TypeRDNAttr {-id id1 -entityTypeName name1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPEntity TypeRDNAttr ('[-id id1 -entityTypeName name1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-entityTypeName', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPEntity TypeRDNAttr {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPEntity TypeRDNAttr (['-interactive'])
```

getIdMgrLDAPGroupConfig

The **getIdMgr LDAPG roupConfig** command returns the LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPGroup Config {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPGroup Config ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPGroup Config (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPGroup Config {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPGroup Config ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPGroup Config (['-interactive'])
```

getIdMgrLDAPGroupDynamicMemberAttrs

The **getIdMgr LDAPGroup Dynamic Member Attrs** command returns the dynamic member attribute configuration from the LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPGroupDynamic MemberAttrs {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPGroupDynamic MemberAttrs ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPGroupDynamic MemberAttrs (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPGroup DynamicMemberAttrs {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPGroup DynamicMemberAttrs ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPGroup DynamicMemberAttrs (['-interactive'])
```

getIdMgrLDAPGroupMemberAttrs

The **getIdMgr LDAPGroup MemberAttrs** command returns the member attribute configuration for the LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPGroup MemberAttrs {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPGroup MemberAttrs ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPGroup MemberAttrs (['-interactive'])
```

getIdMgrLDAPSearchResultCache

The **getIdMgr LDAPSearch ResultCache** command returns the LDAP search result cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAP SearchResultCache {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAP SearchResultCache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPSear chResultCache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPSea rchResultCache {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPSear chResultCache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPSe archResultCache (['-interactive'])
```

getIdMgrLDAPServer

The **getIdMgr LDAPServer** command returns the configuration for the LDAP server that you specify for the LDAP repository ID that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

-host
The host name for the primary LDAP server. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPServer {-id id1 -host  myhost.ibm.com}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPServer ('[-id id1 -host  myhost.ibm.com]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrLDAPServer {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrLDAPServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrLDAPServer (['-interactive'])
```

getIdMgrRepository

The **getIdMgr Repository** command returns the configuration of the specified repository.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrRepos itory {-id id1}
```

- Using Jython string:

```
AdminTask.getIdMgrReposi tory ('[-id id1]')
```

- Using Jython list:

```
AdminTask.getIdMgrReposi tory (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrRepos itory {-interactive}
```

- Using Jython string:

```
AdminTask.getIdMgrReposi tory ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getIdMgrReposi tory (['-interactive'])
```

listIdMgrLDAPAttrs

The listIdMgrLDAPAttrs command lists the name of each configured attributes for the LDAP repository of interest.

Required parameters

-id Specifies the unique ID of the repository. (String, required)

Return value

The command returns a list of HashMaps that contains parameters of the addIdMgrLDAPAttr command as keys.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAPAttrs {-id id1}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAPAttrs ('[-id id1]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAPAttrs (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAPAttrs {-interactive}
```

- Using Jython:

```
AdminTask.listIdMgrLDAPAttrs('-interactive')
```

listIdMgrCustomProperties

The **listIdMgr Custom Properties**command returns a list of custom properties for the repository that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrCustomPro perties {-id id1}
```

- Using Jython string:

```
AdminTask.listIdMgrCustom Properties ('[-id id1]')
```

- Using Jython list:

```
AdminTask.listIdMgrCustom Properties (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrCustom Properties {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrCustom Properties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrCustom Properties (['-interactive'])
```

listIdMgrLDAPBackupServers

The **listIdMgr LDAPBack upServers**command returns a list of the backup LDAP server or servers.

Parameters and return values

-id The ID of the repository. (String, required)

-primary_host
The host name for the primary LDAP server. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP BackupServer {-id id1 -primary_host hostname}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAPBack upServer ('[-id id1 -primary_host hostname]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAPBack upServer (['-id', 'id1', '-primary_host', 'hostname'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP BackupServer {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAP BackupServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAP BackupServer (['-interactive'])
```

listIdMgrLDAPEntityTypes

The **listIdMgr LDAPEn tityTypes** command lists the name of all of the configured LDAP entity type definitions.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP EntityType {-id id1}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAP EntityType ('[-id id1]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAPEntity Type (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAP EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAP EntityType (['-interactive'])
```

listIdMgrLDAPServers

The **listIdMgr LDAP Servers** command lists all of the configured primary LDAP servers.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP Servers {-id id1}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAP Servers ('[-id id1]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAP Servers (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrLDAP Servers {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrLDAP Servers ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrLDAP Servers (['-interactive'])
```

listIdMgrRepositories

The **listIdMgr Repositories** command lists names and types of all configured repositories.

Parameters and return values

- Parameters: None
- Returns: A hash map with key as the name of the repository and value as another hash map that includes the following keys:

– repositoryType - The type of repository. For example, File, LDAP, DB, and so on.
– specificRepositoryType - The specific type of repository. For example, LDAP, IDS51, NDS, and so on.
– host - The host name where the repository resides. For File, it is LocalHost and for DB it is dataSourceName.

This command will not return the Property Extension and Entry Mapping repository data.

Examples

Batch mode example usage:

• Using Jacl:

```
$AdminTask listIdMgrRepositories
```

• Using Jython string:

```
AdminTask.listIdMgrRepositories()
```

• Using Jython list:

```
AdminTask.listIdMgrRepositories()
```

Interactive mode example usage:

• Using Jacl:

```
$AdminTask listIdMgrReposi tories {-interactive}
```

• Using Jython string:

```
AdminTask.listIdMgrReposit ories ('[-interactive]')
```

• Using Jython list:

```
AdminTask.listIdMgrReposi tories (['-interactive'])
```

listIdMgrRepositoryBaseEntries

The **listIdMgr Repository BaseEntries** command lists the base entries for a specified repository.

Parameters and return values

-id The ID of the repository. (String, required)

Examples

Batch mode example usage:

• Using Jacl:

```
$AdminTask listIdMgrRepository BaseEntries {-id id1}
```

• Using Jython string:

```
AdminTask.listIdMgrRepository BaseEntries ('[-id id1]')
```

• Using Jython list:

```
AdminTask.listIdMgrRepository BaseEntries (['-id', 'id1'])
```

Interactive mode example usage:

• Using Jacl:

```
$AdminTask listIdMgrRepository BaseEntries {-interactive}
```

• Using Jython string:

```
AdminTask.listIdMgrRepository BaseEntries ('[-interactive]')
```

• Using Jython list:

```
AdminTask.listIdMgrRepository BaseEntries (['-interactive'])
```

listIdMgrSupportedDBTypes

The **listIdMgr Supported DBTypes** command returns a list of supported database types.

Parameters and return values

- Parameters: None
- Returns: A list of supported database types.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupportedDBTypes
```

- Using Jython string:

```
AdminTask.listIdMgrSupportedDBTypes()
```

- Using Jython list:

```
AdminTask.listIdMgrSupportedDBTypes()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupported DBTypes {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrSupported DBTypes ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrSupported DBTypes (['-interactive'])
```

listIdMgrSupportedMessageDigestAlgorithms

The **listIdMgr Supported Message Digest Algorithms** command returns a list of supported message digest algorithms.

Parameters and return values

- Parameters: None
- Returns: A list of supported message digest algorithms.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupported MessageDigestAlgorithms
```

- Using Jython string:

```
AdminTask.listIdMgrSupported MessageDigestAlgorithms()
```

- Using Jython list:

```
AdminTask.listIdMgrSupported MessageDigestAlgorithms()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupportedMes sageDigestAlgorithms {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrSupportedMessage DigestAlgorithms ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrSupportedMessage DigestAlgorithms (['-interactive'])
```

listIdMgrSupportedLDAPServerTypes

The **listIdMgr Supported LDAP ServerTypes** command returns a list of supported LDAP server types.

Parameters and return values

- Parameters: None
- Returns: A list of supported LDAP server types.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupported LDAPServerTypes
```

- Using Jython string:

```
AdminTask.listIdMgrSupported LDAPServerTypes()
```

- Using Jython list:

```
AdminTask.listIdMgrSupported LDAPServerTypes()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listIdMgrSupported LDAPServerTypes {-interactive}
```

- Using Jython string:

```
AdminTask.listIdMgrSupported LDAPServerTypes ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listIdMgrSupported LDAPServerTypes (['-interactive'])
```

removeIdMgrLDAPBackupServer

The **removeIdMgr LDAPBack upServer**command removes the backup LDAP server or servers.

Parameters and return values

-id The ID of the repository. (String, required)

-primary_host
The host name for the primary LDAP server. (String, required)

-host
The name of the backup host name. Use a asterisk (*) if you want to remove all backup servers. (String, required)

-port
The port number of the LDAP server. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeIdMgrLDAP BackupServer {-id id1 -primary_host myprimaryhost. ibm.com -host myhost.ibm.com}
```

- Using Jython string:

```
AdminTask.removeIdMgrLDAPBackup Server ('[-id id1 -primary_host myprimaryhost.ibm.com -host myhost.ibm.com]')
```

- Using Jython list:

```
AdminTask.removeIdMgrLDAPBack upServer (['-id', 'id1', '-primary_host', 'myprimary host.ibm.com', '-host', 'myhost.ibm.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeIdMgrLDAP BackupServer {-interactive}
```

- Using Jython string:

```
AdminTask.removeIdMgrLDAP BackupServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeIdMgrLDAP BackupServer (['-interactive'])
```

setIdMgrCustomProperty

The **setIdMgr Custom Property** command : sets, adds or deletes a custom property to a repository configuration. If a value is not specified, or if there is an empty string, the property is deleted from the repository configuration. If a name does not exist it is added if a value is specified. If the name is ″*″ then all of the custom properties are deleted.

Parameters and return values

-id The unique identifier of the repository. Valid values include the existing repository IDs. (String, required)

-name
The name of the additional property for the repository that are not defined OOTB.(String, required)

-value
The value of a property for the repository. If this parameter is an empty string, the property is deleted from the repository configuration. If this parameter is not an empty string, and a name does not exist, it is added. If a name is an empty string, all of the custom properties are deleted. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrCustomProperty {-id id1 -name name1 -value value}
```

- Using Jython string:

```
AdminTask.setIdMgrCustomProperty ('[-id id1 -name name1 -value value]')
```

- Using Jython list:

```
AdminTask.setIdMgrCustomProperty (['-id', 'id1', '-name', 'name1', '-value', 'value'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrCustom Property {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrCustom Property ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrCustom Property (['-interactive'])
```

setIdMgrLDAPAttrCache

The **setIdMgr LDAPA ttrCache** command configures the LDAP attribute cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-cachesDiskOffLoad
(String, optional)

-enabled
Indicates if you want to enable attribute caching. The default value is `true`. (Boolean, optional)

-cacheSize
The maximum size of the attribute cache defined by the number of attribute objects that are permitted in the attribute cache. The minimum value of this parameter is `100`. The default value is `4000`. (Integer, optional)

-cacheTimeOut
The amount of time in seconds before the cached entries that are located in the attributes cache can be not valid. The minimum value of this parameter is `0`. The attribute objects that are cached will remain in the attributes cache until the virtual member manager changes the attribute objects. The default value is `1200`. (Integer, optional)

-attributeSizeLimit
An integer that represents the maximum number of attribute object values that can cache in the attributes cache.

Some attributes, for example, the member attribute, contain many values. The attributeSizeLimit parameter prevents the attributes cache to cache large attributes. The default value is `2000`. (Integer, optional)

-serverTTLAttribute
The name of the ttl attribute that is supported by the LDAP server. The attributes cache uses the value of this attribute to determine when the cached entries in the attributes cache will time out.

The ttl attribute contains the time, in seconds, that any information from the entry should be kept by a client before it is considered stale and a new copy is fetched. A value of `0` implies that the object will not be cached. For more information about this attribute, go to: http://www.ietf.org/proceedings/98aug/I-D/draft-ietf-asid-ldap-cache-01.txt.

The ttl attribute is not supported by all LDAP servers. If this attribute is supported by an LDAP server, you can set the value of the serverTTLAttribute parameter to the name of the ttl attribute in order to allow the value of the ttyl attribute to determine when cached entries will time out. The time out value for different entries in attributes cache can be different.

For example, if the value of the serverTTLAttribute parameter is ttl and the attributes cache retrieves attributes of a user from an LDAP server, it will also retrieve the value of the ttl attribute of this user. If the value is 200, the WMM uses this value to set the time out for the attributes of the user in the attributes cache instead of using the value of cacheTimeout. You can set different ttl attribute values for different users. (String, optional)

- Returns: None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPAttr Cache {-id id1}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPAttr Cache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPAttr Cache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPAttr Cache {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPAttr Cache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPAttr Cache (['-interactive'])
```

setIdMgrLDAPContextPool

The **setIdMgr LDAPCo ntextPool**command sets up the LDAP context pool configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-enabled
By default, the context pool is enabled. If you set this parameter to `false`, the context pool is disabled. When the context pool is disabled, new context instances will be created for each request. The default value is `true`. (Boolean, optional)

-initPoolSize
The number of context instances that the virtual member manager LDAP adapter creates when it creates the pool. The valid range for this parameter is `1` to `50`. The default value is `1`. (Integer, optional)

-maxPoolSize
The maximum number of context instances that the context pool will maintain. Context instances that are in use and those that are idle contribute to this number. When the pool size reaches this number, new context instances cannot be created for new requests. The new request is blocked until a context instance is released by another request or is removed. The request checks periodically if there are context instances available in the pool according to the amount of time that you specify using the poolWaitTime parameter.

The minimum value for this parameter is `0`. There is no maximum value. Setting the value of this parameter to `0` means that there is no maximum size and a request for a pooled context instance will use an existing pooled idle context instance or a newly created pooled context instance. The default value is `20`.(Integer, optional)

-prefPoolSize
The preferred number of context instances that the context pool will maintain. Context instances that are in use and those that are idle contribute to this number. When there is a request for the use of a pooled context instance and the pool size is less than the preferred size, the context pool creates and uses a new pooled context instance regardless of whether an idle connection is available. When a request finishes with a pooled context instance and the pool size is greater than the preferred size, the context pool closes and removes the pooled context instance from the pool.

The valid range for this parameter is from `0` to `100`. Setting the value of this parameter to `0` means that there is no preferred size and a request for a pooled context instance results in a newly created context instance only if no idle ones are available. The default value is `3`.(Integer, optional)

-poolTimeOut
An integer that represents the number of milliseconds that an idle context instance may remain in the pool without being closed and removed from the pool. When a context instance is requested from the pool, if this context already exists in the pool for more than the time defined by poolTimeout, this connection will be closed no matter this context instance is stale or active. A new context instance will be created and put back to the pool after it has been released from the request.

The minimum value for this parameter is `0`. There is no maximum value. Setting the value of this parameter to `0` means that the context instances in the pool will remain in the pool until they are staled. The context pool catches the communication exception and recreates a new context instance. The default value is `0`.(Integer, optional)

-poolWaitTime
The time interval in milliseconds that the request waits until the context pool rechecks if there are idle

context instances available in the pool when the number of context instances reaches the maximum pool size. If no idle context instance, the request will continue waiting for the same period of time until next checking.

The minimum value for the poolWaitout parameter is `0`. There is no maximum value. A value of `0` for this parameter means that the context pool will not check if idle context exists. The request will be notified when a context instance releases from other requests. The default value is `3000`.(Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPCon textPool {-id id1}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPCon textPool ('[-id id1]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPCon textPool (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPCon textPool {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPCon textPool ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPCon textPool (['-interactive'])
```

setIdMgrLDAPGroupConfig

The **setIdMgr LDAPGr oupConfig** command sets up the LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-updateGroup Membership
Updates the group membership if the member is deleted or renamed. Some LDAP servers, for example, Domino server, do not clean up the membership of the user when a user is deleted or renamed. If you choose these LDAP server types in the ldapServerType property, the value of this parameter is set to `true`. Use this parameter to change the value. The default value is `false`. (Boolean, optional)

-name
The name of the membership attribute. For example, `memberOf` in an active directory server and `ibm-allGroups` in IDS. (String, optional)

-scope
The scope of the membership attribute. The following are the possible values for this parameter:

- direct - The membership attribute only contains direct groups. Direct groups contain the member and are not contained through a nested group. For example, if group1 contains group2, group2 contains user1, then group2 is a direct group of user1, but group1 is not a direct group of user1.
- nested - The membership attribute contains both direct groups and nested groups.
- all - The membership attribute contains direct groups, nested groups, and dynamic members.

The default value is `direct`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAP GroupConfig {-id id1}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAP GroupConfig ('[-id id1]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAP GroupConfig (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPGr oupConfig {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPGr oupConfig ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPGroup Config (['-interactive'])
```

setIdMgrLDAPSearchResultCache

The **setIdMgr LDAPSearch ResultCache** command sets up the LDAP search result cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-cachesDiskOffLoad
Loads the attributes caches and the search results onto hard disk. By default, when the number of cache entries reaches the maximum size of the cache, cache entries are evicted to allow new entries to enter the caches. If you enable this parameter, the evicted cache entries will be copied to disk for future access. The default value is `false`. (Boolean, optional)

-enabled
Enables the search results cache. The default value is `true`. (Boolean, optional)

-cacheSize
The maximum size of the search results cache. The number of naming enumeration objects that can be put into the search results cache. The minimum value of this parameter is `100`. The default value is `2000`. (Integer, optional)

-cacheTimeOut
The amount of time in seconds before the cached entries in the search results cache can be not valid. The minimum value for this parameter is `0`. A value of `0` means that the cached naming enumeration objects will stay in the search results cache until there are configuration changes. The default value is `600`. (Integer, optional)

-searchResultSizeLimit
The maximum number of entries contained in the naming enumeration object that can be cached in the search results cache.For example, if the results from a search contains 2000 users, the search results will not cache in the search results cache if the value of the of this property is set to `1000`. The default value is `1000`. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPSearch ResultCache {-id id1}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPSearch ResultCache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPSearch ResultCache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrLDAPSearch ResultCache {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrLDAPSearch ResultCache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrLDAPSearch ResultCache (['-interactive'])
```

setIdMgrEntryMappingRepository

The **setIdMgr Entry Mapping Repository** command sets or updates an entry mapping repository configuration.

Parameters and return values

-dataSourceName
The name of the data source. The default value is `jdbc/wimDS`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-databaseType
The type of the database. The default value is `DB2`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-dbURL
The URL of the database. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-dbAdminId
The database administrator ID. (String, required if database type is not Apache Derby.)

-dbAdminPassword
The database administrator password. (String, required if database type is not Apache Derby.)

-JDBCDriverClass
The JDBC driver class name. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrEntry MappingRepository {-dbAdminId dbid1 -dbAdminPassword pw1}
```

- Using Jython string:

```
AdminTask.setIdMgrEntry MappingRepository ('[-dbAdminId dbid1 -dbAdminPassword pw1]')
```

- Using Jython list:

```
AdminTask.setIdMgrEntry MappingRepository (['-dbAdminId', 'dbid1', '-dbAdmin Password', 'pw1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrEntryMapping Repository {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrEntryMapping Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrEntryMapping Repository (['-interactive'])
```

setIdMgrPropertyExtensionRepository

The **setIdMgr Property Extension Repository** command sets or updates the property extension repository configuration.

Parameters and return values

-dataSourceName
The name of the data source. The default value is `jdbc/wimDS`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-databaseType
The type of the database. The default value is `DB2`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-dbURL
The URL of the database. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)

-dbAdminId
The database administrator ID. (String, required if database type is not Apache Derby.)

-dbAdminPassword
The database administrator password. (String, required if database type is not Apache Derby.)

-entityRetrievalLimit
The limit for the retrieval of entities. (Integer, required)

-JDBCDriverClass
The JDBC driver class name. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrProperty ExtensionRepository {-entity RetrievalLimit 10 -JDBC DriverClass classname}
```

- Using Jython string:

```
AdminTask.setIdMgrProperty ExtensionRepository ('[-entity RetrievalLimit 10 -JDBC DriverClass classname]')
```

- Using Jython list:

```
AdminTask.setIdMgrProperty ExtensionRepository (['-entity RetrievalLimit', '10', '-JDBCDriverClass', 'classname'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrProperty ExtensionRepository {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrProperty ExtensionRepository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setIdMgrPropertyExt ensionRepository (['-interactive'])
```

updateIdMgrDBRepository

The **updateId MgrDB Repository** command updates the configuration for the database repository that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

-dataSourceName
The name of the data source. The default value is `jdbc/wimDS`. (String, optional)

-databaseType
The type of the database. The default value is `DB2`. (String, optional)

-dbURL
The URL of the database. (String, optional)

-dbAdminId
The database administrator ID. (String, optional)

-dbAdminPassword
The database administrator password. (String, optional)

-entityRetrievalLimit
Indicates the value of the retrieval limit on database entries. The default value is `200`. (Integer, optional)

-JDBCDriverClass
The JDBC driver class name. (String, optional)

-saltLength
The salt length in bits. The default value is `12`. (Integer, optional)

-encryptionKey
The default value is `rZ15ws0ely9yHk3zCs3sTMv/ho8fY17s`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrDB Repository {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrDB Repository ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrDB Repository (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrDB Repository {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrDB Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrDB Repository (['-interactive'])
```

updateIdMgrFileRepository

The **updateId MgrFile Repository** command updates the configuration for the file repository that you specify. To update other properties of the file repository use the **update IdMgr Repository**command.

Parameters and return values

-id The ID of the repository. (String, required)

-messageDigest Algorithm
The message digest algorithm that will be used for hashing the password. The default value is `SHA-1`. Valid values include the following: `SHA-1`, `SHA-384`, or `SHA-512`.(String, optional)

-baseDirectory
The base directory where the fill will be created in order to store the data. The default is to be dynamically built during run time using user.install.root and cell name. (String, optional)

-fileName
The file name of the repository. The default value is `fileRegistry.xml`. (String, optional)

-saltLength
The salt length of the randomly generated salt for password hashing. The default value is `12`. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrFile Repository {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrFile Repository ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrFile Repository (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrFile Repository {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrFile Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrFile Repository (['-interactive'])
```

updateIdMgrLDAPAttrCache

The **updateId MgrLDAP AttrCache** command updates the LDAP attribute cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-cachesDiskOffLoad
(String, optional)

-enabled
Indicates if you want to enable attribute caching. The default value is `true`. (Boolean, optional)

-cacheSize
The maximum size of the attribute cache defined by the number of attribute objects that are permitted in the attribute cache. The minimum value of this parameter is `100`. The default value is `4000`. (Integer, optional)

-cacheTimeOut
The amount of time in seconds before the cached entries that are located in the attributes cache can

be not valid. The minimum value of this parameter is 0. The attribute objects that are cached will remain in the attributes cache until the virtual member manager changes the attribute objects. The default value is 1200. (Integer, optional)

-attributeSizeLimit
An integer that represents the maximum number of attribute object values that can cache in the attributes cache.

Some attributes, for example, the member attribute, contain many values. The attributeSizeLimit parameter prevents the attributes cache to cache large attributes. The default value is 2000. (Integer, optional)

-serverTTLAttribute
The name of the ttl attribute that is supported by the LDAP server. The attributes cache uses the value of this attribute to determine when the cached entries in the attributes cache will time out.

The ttl attribute contains the time, in seconds, that any information from the entry should be kept by a client before it is considered stale and a new copy is fetched. A value of 0 implies that the object will not be cached. For more information about this attribute, go to: http://www.ietf.org/proceedings/98aug/I-D/draft-ietf-asid-ldap-cache-01.txt.

The ttl attribute is not supported by all LDAP servers. If this attribute is supported by an LDAP server, you can set the value of the serverTTLAttribute parameter to the name of the ttl attribute in order to allow the value of the ttyl attribute to determine when cached entries will time out. The time out value for different entries in attributes cache can be different.

For example, if the value of the serverTTLAttribute parameter is ttl and the attributes cache retrieves attributes of a user from an LDAP server, it will also retrieve the value of the ttl attribute of this user. If the value is 200, the WMM uses this value to set the time out for the attributes of the user in the attributes cache instead of using the value of cacheTimeout. You can set different ttl attribute values for different users.

(String, optional)

- Returns: None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP AttrCache {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP AttrCache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP AttrCache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP AttrCache {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP AttrCache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP AttrCache (['-interactive'])
```

updateIdMgrLDAPContextPool

The **updateId MgrLDAP ContextPool** command updates the LDAP context pool configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-enabled
By default, the context pool is enabled. If you set the value of this parameter to `false`, the context pool is disabled which means that a new context instance will be created for each request. The default value is `true`. (Boolean, optional)

-initPoolSize
The number of context instances that the virtual member manager LDAP adapter creates when it creates the pool. The valid range for this parameter is `1` to `50`. The default value is `1`. (Integer, optional)

-maxPoolSize
The maximum number of context instances that can be maintained concurrently by the context pool. Both in-use and idle context instances contribute to this number. When the pool size reaches this number, new context instances cannot created for new request. The new request is blocked until a context instance is released by another request or is removed. The request checks periodically if there are context instances available in the pool according to the value defined for the poolWaitTime parameter. The minimum value of the maxPoolSize parameter is `0`. There is no maximum value. A maximum pool size of 0 means that there is no maximum size and that a request for a pooled context instance will use an existing pooled idle context instance or a newly created pooled context instance. The default value is `20`. (Integer, optional)

-prefPoolSize
The preferred number of context instances that the Context Pool should maintain. Both in-use and idle context instances contribute to this number. When there is a request for the use of a pooled context instance and the pool size is less than the preferred size, Context Pool will create and use a new pooled context instance regardless of whether an idle connection is available. When a request is finished with a pooled context instance and the pool size is greater than the preferred size, the Context Pool will close and remove the pooled context instance from the pool. The valid range of the prefPoolSize parameter is `0` to `100`. A preferred pool size of 0 means that there is no preferred size: A request for a pooled context instance will result in a newly created context instance only if no idle ones are available. The default value is `3`. (Integer, optional)

-poolTimeOut
An integer that represents the number of milliseconds that an idle context instance may remain in the pool without being closed and removed from the pool. When a context instance is requested from the pool, if this context already exists in the pool for more than the time defined by poolTimeout, this connection will be closed no matter this context instance is stale or active. A new context instance will be created and put back to the pool after it has been released from the request.The minimum value of poolTimeout is 0. There is no maximum value.A poolTimeout of 0 means that the context instances in the pool will remain in the pool until they are staled. In this case, Context Pool will catch the communication exception and recreate a new context instance. The default value is `0`. (Integer, optional)

-poolWaitTime
The time interval (in milliseconds) that the request will wait until the Context Pool checks again if there are idle context instance available in the pool when the number of context instances reaches the maximum pool size. If there is still no idle context instance, the request will continue waiting for the same period of time until next checking. The minimum value of poolWaitout is 0. There is no maximum value. A poolWaitTime of 0 means the Context Pool will not check if there are idle context. Instead, the request will be notified when there is a context instance is released from other requests. The default value is `3000`. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP ContextPool {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP ContextPool ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP ContextPool (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP ContextPool {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP ContextPool ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP ContextPool (['-interactive'])
```

updateIdMgrLDAPEntityType

The **updateId MgrLDAP EntityType** command updates an existing LDAP entity type definition to LDAP repository configuration. You can use this command to add more values to multi-valued parameters. If the property already exists, the value of the property will be replaced. If the property does not exist, it will be added.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the entity type. (String, required)

-searchFilter
The search filter that you want to use to search the entity type. (String, optional)

-objectClasses
One or more object classes for the entity type. (String, optional)

-objectClassesForCreate
The object class that will be when you create an entity type object. You do not have to specify the value of this parameter if it is the same as the value of the objectClasses parameter. (String, optional)

-searchBases
The search base or bases to use while searching the entity type. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAPEntity Type {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPEntity Type ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP EntityType {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP EntityType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP EntityType (['-interactive'])
```

updateIdMgrLDAPGroupDynamicMemberAttr

The **updateIdMgr LDAPGroup Dynamic MemberAttr** command updates a dynamic member attribute configuration to an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the LDAP attribute that is used as the group member attribute. For example, `memberURL`. (String, required)

-objectClass
The group object class that contains the dynamic member attribute. For example `groupOfURLs`. If you do not define this parameter, the dynamic member attribute will apply to all group object classes. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAPGroup DynamicMemberAttr {-id id1 -name name1 -objectClass groupOfURLs}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr ('[-id id1 -name name1 -objectClass groupOfURLs]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr (['-id', 'id1', '-name', 'name1', '-objectClass', 'groupOfURLs'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAPGroup DynamicMemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])
```

updateIdMgrLDAPGroupMemberAttr

The **updateIdMgr LDAPGroup MemberAttr** command updates a member attribute configuration of an LDAP group configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required)

-objectClass
The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUnqiueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)

-scope
The scope of the member attribute. The following are the valid values:

- direct - The member attribute only contains direct members whereby the member is directly contained by the group and not contained in a nested group. For example, if group1 contains group2, group2 contains user1, then group2 is a direct member of group1 but user1 is not a direct member of group1. Both member and uniqueMember are direct member attributes.
- nested - The member attribute contains both direct members and nested members.

-dummyMember
When you create a group without specifying a member, a dummy member will be filled in automatically to avoid receiving an exception that indicates that there is a mandatory attribute missing. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP GroupMemberAttr {-id id1 -name name1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP GroupMemberAttr ('[-id id1 -name name1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP GroupMemberAttr (['-id', 'id1', '-name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAPGroup MemberAttr {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPGroup MemberAttr ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPGroup MemberAttr (['-interactive'])
```

updateIdMgrLDAPRepository

The **updateId MgrLDAP Repository** command updates an LDAP repository configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-ldapServerType
The type of LDAP server that is being used. The default value is `IDS51`. (String, optional)

-adapterClassName
The default value is `com.ibm.ws. wim.adapter. ldap.LdapA dapter`. (String, optional)

-certificateMapMode
Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `exactdn`. To use the certificate filter for the mapping, specify `certificatefilter`. (String, optional)

-certificateFilter
If `certificateMapMode` has the value `certificatefilter`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional)

-isExtIdUnique
Specifies if the external ID is unique. The default value is `true`. (Boolean, optional)

-loginProperties
Indicates the property name used for login. (String , optional)

-primaryServerQueryTimeInterval
Indicates the polling interval for testing the primary server availability. The value of this parameter is specified in minutes. The default value is `15`. (Integer, optional)

-returnToPrimaryServer
Indicates to return to the primary LDAP server when it is available. The default value is `true`. (Boolean, optional)

-supportAsyncMode
Indicates if the async mode is supported or not. The default value is `false`. (Boolean, optional)

-supportSorting
Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional)

-supportPaging
Indicates if paging is supported or not. The default value is `false`. (Boolean, optional)

-supportTransactions
Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional)

-supportExternalName
Indicates if external names are supported or not. The default value is `false`. (Boolean, optional)

-sslConfiguration
The SSL configuration. (String, optional)

-translateRDN
Indicates to translate RDN or not. The default value is `false`. (Boolean, optional)

-searchTimeLimit
The value of search time limit. (Integer, optional)

-searchCountLimit
The value of search count limit. (Integer, optional)

-searchPageSize
The value of search page size. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP Repository {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP Repository ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP Repository (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP Repository {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP Repository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP Repository (['-interactive'])
```

updateIdMgrLDAPSearchResultCache

The **updateIdMgr LDAPSearch ResultCache** command updates the LDAP search result cache configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-cachesDiskOffLoad
Loads the attributes caches and the search results onto hard disk. By default, when the number of cache entries reaches the maximum size of the cache, cache entries are evicted to allow new entries to enter the caches. If you enable this parameter, the evicted cache entries will be copied to disk for future access. The default value is `false`. (Boolean, optional)

-enabled
Enables the search results cache. The default value is `true`. (Boolean, optional)

-cacheSize
The maximum size of the search results cache. The number of naming enumeration objects that can be put into the search results cache. The minimum value of this parameter is `100`. The default value is `2000`. (Integer, optional)

-cacheTimeOut
The amount of time in seconds before the cached entries in the search results cache can be not valid. The minimum value for this parameter is `0`. A value of `0` means that the cached naming enumeration objects will stay in the search results cache until there are configuration changes. The default value is `600`. (Integer, optional)

-searchResultSizeLimit
The maximum number of entries contained in the naming enumeration object that can be cached in the search results cache.For example, if the results from a search contains 2000 users, the search results will not cache in the search results cache if the value of the of this property is set to `1000`. The default value is `1000`. (Integer, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP SearchResultCache {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPSearch ResultCache ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPSearch ResultCache (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP SearchResultCache {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPSearch ResultCache ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPSearch ResultCache (['-interactive'])
```

updateIdMgrLDAPServer

The **updateIdMgr LDAPServer**command updates an LDAP server configuration for the LDAP repository ID that you specify.

Parameters and return values

-id The ID of the repository. (String, required)

-host
The host name for the LDAP server that contains the properties that you want to modify. (String, required)

-port
The port number for the LDAP server. (Integer, optional)

-authentication
Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional)

-bindDN
The binding domain name for the LDAP server. (String, optional)

-bindPassword
The binding password. The password is encrypted before it is stored.(String, optional)

-certificateMapMode
Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `exactdn`. To use the certificate filter for the mapping, specify `certificatefilter`. (String, optional)

-certificateFilter
If `certificateMapMode` has the value `certificatefilter`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional)

-connectTimeout
The connection timeout measured in seconds. The default value is `0`. (Integer, optional)

-connectionPool
The connection pool. The default value is `false`. (Boolean, optional)

-derefAliases
Controls how aliases are dereferenced. The default value is `always`. Valid values include:

- `never` - never deference aliases
- `finding` - deferences aliases only during name resolution
- `searching` - deferences aliases only after name resolution

(String, optional)

-ldapServerType
The type of LDAP server being used. The default value is `IDS51`. (String, optional)

-primary_host
The host name for the primary LDAP server. (String, optional)

-referal
The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional)

-sslConfiguration
The SSL configuration. (String, optional)

-sslEnabled
Indicates to enable SSL or not. The default value is `false`. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAPServer {-id id1 -host myhost.ibm.com}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAPServer ('[-id id1 -host myhost.ibm.com]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrLDAP Server {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrLDAP Server ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrLDAP Server (['-interactive'])
```

updateIdMgrRepository

The **updateIdMgr Repository**command updates the common repository configuration.

Parameters and return values

-id The ID of the repository. (String, required)

-adapterClassName
The implementation class name for the repository adapter. (String, optional)

-EntityTypesNot AllowCreate
The name of the entity type that should not be created in this repository. (String, optional)

-EntityTypesNotAllowUpdate
The name of the entity type that should not be updated in this repository. (String, optional)

-EntityTypesNotAllowRead
The name of the entity type that should not be read from this repository. (String, optional)

-EntityTypesNotAllowDelete
The name of the entity type that should not be deleted from this repository. (String, optional)

-loginProperties
(String, optional)

-readOnly
Indicates if this is a read only repository. The default value is `false`. (Boolean, optional)

-repositoriesForGroups
The repository ID where group data is stored. (String, optional)

-supportPaging
Indicates if the repository supports paging or not. (Boolean, optional)

-supportSorting
Indicates if the repository supports sorting or not. (Boolean, optional)

-supportTransactions
Indicates if the repository supports transaction or not. (Boolean, optional)

-isExtIdUnique
Specifies if the external ID is unique or not. (Boolean, optional)

-supportedExternalName
Indicates if the repository supports external names or not. (Boolean, optional)

-supportAsyncMode
Indicates if the adapter supports async mode or not. The default value is `false`. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrRe pository {-id id1}
```

- Using Jython string:

```
AdminTask.updateIdMgrRe pository ('[-id id1]')
```

- Using Jython list:

```
AdminTask.updateIdMgr Repository (['-id', 'id1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrRe pository {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrRe pository ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrRe pository (['-interactive'])
```

updateIdMgrRepositoryBaseEntry

The **updateIdMgr Repository BaseEntry** command updates a base entry to the specified repository.

Parameters and return values

-id The ID of the repository. (String, required)

-name
The distinguished name of a base entry. (String, required)

-nameInRepository
The distinguished name in the repository that uniquely identifies the base entry name. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrReposi toryBaseEntry {-id id1 name name1}
```

- Using Jython string:

```
AdminTask.updateIdMgrRepository BaseEntry ('[-id id1 name name1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrReposi toryBaseEntry (['-id', 'id1', 'name', 'name1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrRepositoryBaseEntry {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrRepositoryBaseEntry ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrRepositoryBaseEntry (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

IdMgrRealmConfig command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure the member manager realm and realms. The commands and parameters in the IdMgrRealmConfig group can be used to create and manage your realm configuration.

The IdMgrRealmConfig command group for the AdminTask object includes the following commands:

addIdMgrRealmBaseEntry

The **addIdMgrRealmBaseEntry** command adds a base entry to a specific realm configuration and links the realm with the repository.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

-baseEntry
Specifies the name of the base entry. (String, optional)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrRealmBaseEntry {-name defaultWIMFileBasedRealm -baseEntry o=sampleFileRepository}
```

- Using Jython string:

```
AdminTask.addIdMgrRealmBaseEntry ('[-name defaultWIMFileBasedRealm -baseEntry o=sampleFileRepository]')
```

- Using Jython list:

```
AdminTask.addIdMgrRealmBaseEntry (['-name', 'defaultWIMFileBasedRealm', '-baseEntry', 'o=sampleFileRepository'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask addIdMgrRealmBaseEntry {-interactive}
```

- Using Jython string:

```
AdminTask.addIdMgrRealmBaseEntry ('[-interactive]')
```

createIdMgrRealm

The **createIdMgrRealm** command creates a realm configuration.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Optional parameters

-securityUse
Specifies a string that indicates if this virtual realm will be used in security now, later, or never. The default value is `active`. Additional values includes: `inactive` and `nonSelectable`. (String, optional)

-delimiter
Specifies the delimiter used for this realm. The default value is `/`. (String, optional)

-allowOperationIfReposDown
Specifies whether the system allows a repository operation such as `get` or `search` to complete successfully, even if repositories in the realm are down. The default value is `false`. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrRealm {-name realm1 -allowOperationIfReposDown true}
```

- Using Jython string:

```
AdminTask.createIdMgrRealm ('[-name realm1 -allowOperationIfReposDown true]')
```

- Using Jython list:

```
AdminTask.createIdMgrRealm (['-name', 'realm1', '-allowOperationIfReposDown', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createIdMgrRealm {-interactive}
```

- Using Jython string:

```
AdminTask.createIdMgrRealm ('[-interactive]')
```

deleteIdMgrRealm

The **deleteIdMgrRealm** command deletes the realm configuration that you specified.

Target Object

None.

Required parameters

-name
The realm name. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteIdMgrRealm {-name realm1}
  ```
- Using Jython string:
  ```
  AdminTask.deleteIdMgrRealm ('[-name realm1]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteIdMgrRealm (['-name', 'realm1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteIdMgrRealm {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.deleteIdMgrRealm ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteIdMgrRealm (['-interactive'])
  ```

deleteIdMgrRealmBaseEntry

The **deleteIdMgrRealmBaseEntry** command deletes a base entry from a realm configuration that you specified.

The realm must always contain at least one base entry, thus you cannot remove every entry.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

-baseEntry
Specifies the name of a base entry. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteIdMgrRealmBaseEntry {-name realm1 -baseEntry entry1}
  ```
- Using Jython string:
  ```
  AdminTask.deleteIdMgrRealmBaseEntry ('[-name realm1 -baseEntry entry1]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteIdMgrRealmBaseEntry (['-name', 'realm1', '-baseEntry', 'entry1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteIdMgrRealmBaseEntry {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.deleteIdMgrRealmBaseEntry ('[-interactive]')
  ```

getIdMgrDefaultRealm

The **getIdMgrDefaultRealm** command returns the default realm name.

Target Object

None.

Required parameters

None.

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getIdMgrDefaultRealm
  ```
- Using Jython string:
  ```
  AdminTask.getIdMgrDefaultRealm()
  ```
- Using Jython list:
  ```
  AdminTask.getIdMgrDefaultRealm()
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getIdMgrDefaultRealm {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getIdMgrDefaultRealm ('[-interactive]')
  ```

getIdMgrRepositoriesForRealm

The **getIdMgrRepositoriesForRealm** command returns repository specific details for the repositories configured for a specified realm.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getIdMgrRepositoriesForRealm {-name realm1}
  ```
- Using Jython string:
  ```
  AdminTask.getIdMgrRepositoriesForRealm ('[-name realm1]')
  ```
- Using Jython list:
  ```
  AdminTask.getIdMgrRepositoriesForRealm (['-name', 'realm1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getIdMgrRepositoriesForRealm {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getIdMgrRepositoriesForRealm ('[-interactive]')
  ```

getIdMgrRealm

The **getIdMgrRealm** command returns the configuration parameters for the realm that you specified.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getIdMgrRealm {-name realm1}
```
- Using Jython string:
```
AdminTask.getIdMgrRealm ('[-name realm1]')
```
- Using Jython list:
```
AdminTask.getIdMgrRealm (['-name', 'realm1'])
```

Interactive mode example usage:
- Using Jacl:
```
$AdminTask getIdMgrRealm {-interactive}
```
- Using Jython string:
```
AdminTask.getIdMgrRealm ('[-interactive]')
```

listIdMgrRealms

The **listIdMgrRealms** command returns all of the names of the configured realms.

Target Object

None.

Required parameters

None.

Optional parameters

None.

Examples

Batch mode example usage:
- Using Jacl:
```
$AdminTask listIdMgrRealms
```
- Using Jython string:
```
AdminTask.listIdMgrRealms()
```
- Using Jython list:
```
AdminTask.listIdMgrRealms()
```

Interactive mode example usage:
- Using Jacl:
```
$AdminTask listIdMgrRealms {-interactive}
```
- Using Jython string:
```
AdminTask.listIdMgrRealms ('[-interactive]')
```

listIdMgrRealmBaseEntries

The **listIdMgrRealmBaseEntries** command returns all of the names of the configured realms.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  `$AdminTask listIdMgrRealmBaseEntries {-name realm1}`
- Using Jython string:
  `AdminTask.listIdMgrRealmBaseEntries ('[-name realm1]')`
- Using Jython list:
  `AdminTask.listIdMgrRealmBaseEntries (['-name', 'realm1'])`

Interactive mode example usage:

- Using Jacl:
  `$AdminTask listIdMgrRealmBaseEntries {-interactive}`
- Using Jython string:
  `AdminTask.listIdMgrRealmBaseEntries ('[-interactive]')`

renameIdMgrRealm

The **renameIdMgrRealm** command renames the name of the realm that you specified.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:
  `$AdminTask renameIdMgrRealm {-name realm1}`
- Using Jython string:
  `AdminTask.renameIdMgrRealm ('[-name realm1]')`
- Using Jython list:
  `AdminTask.renameIdMgrRealm (['-name', 'realm1'])`

Interactive mode example usage:

- Using Jacl:
  `$AdminTask renameIdMgrRealm {-interactive}`
- Using Jython string:
  `AdminTask.renameIdMgrRealm ('[-interactive]')`

- Using Jython list:

```
AdminTask.renameIdMgrRealm (['-interactive'])
```

setIdMgrDefaultRealm

The **setIdMgrDefaultRealm** command sets up the default realm configuration.

Parameters and return values

-name
specifies the name of the realm that is used as a default realm when the caller does not specify any in context. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrDefaultRealm {-name realm1}
```

- Using Jython string:

```
AdminTask.setIdMgrDefaultRealm ('[-name realm1]')
```

- Using Jython list:

```
AdminTask.setIdMgrDefaultRealm (['-name', 'realm1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setIdMgrDefaultRealm {-interactive}
```

- Using Jython string:

```
AdminTask.setIdMgrDefaultRealm ('[-interactive]')
```

updateIdMgrRealm

The **updateIdMgrRealm** command updates the configuration for a realm that you specify.

Target Object

None.

Required parameters

-name
Specifies the name of the realm. (String, required)

Optional parameters

-securityUse
Specifies a string that indicates if this realm will be used in security now, later, or never. The default value is `active`. Additional values includes: `inactive` and `nonSelectable`. (String, optional)

-delimiter
specifies the delimiter used for this realm. The default value is /. (String, optional)

-allowOperationIfReposDown
Specifies whether the system allows a repository operation such as `get` or `search` to complete successfully, even if repositories in the realm are down. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrRealm {-name realm1}
```

- Using Jython string:

```
AdminTask.updateIdMgrRealm ('[-name realm1]')
```

- Using Jython list:

```
AdminTask.updateIdMgrRealm (['-name', 'realm1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateIdMgrRealm {-interactive}
```

- Using Jython string:

```
AdminTask.updateIdMgrRealm ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateIdMgrRealm (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

"IdMgrRepositoryConfig command group for the AdminTask object" on page 835
You can use the Jython or Jacl scripting languages to configure security. The commands and parameters in the IdMgrRepositoryConfig group can be used to create and manage the virtual member manager and LDAP directory properties.

"IdMgrConfig command group for the AdminTask object" on page 830
You can use the Jython or Jacl scripting languages to configure the virtual member manager with the wsadmin tool. The commands and parameters in the IdMgrConfig group can be used to create and manage your entity type configuration.

WIMManagementCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the WIMManagementCommands group can be used to create and manage groups, members, and users in the virtual member manager.

The WIMManagementCommands command group for the AdminTask object includes the following commands:

createGroup

The **createGroup** command creates a new group in the virtual member manager. After the command completes, the new group will appear in the repository. For LDAP, a group must contain a member. The memberUniqueName parameter is optional in this case. If you set the memberUniqueName parameter to the unique name of a group or a user, the group or user will be added as a member of the group.

Parameters and return values

-cn
Specifies the common name for the group that you want to create. This parameter maps to the cn property in virtual member manager. (String, required)

-description
Specifies additional information about the group that you want to create. This parameter maps to the description property in a virtual member manager object. (String, optional)

-parent
Specifies the repository in which you want to create the group. This parameter maps to the parent property in the virtual member manager. (String, optional)

-memberUniqueName
Specifies the unique name value for the user or group that you want to add to the new group. This parameter maps to the uniqueName property in the virtual member manager. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createGroup {-cn groupA -description a group of admins}
```

- Using Jython string:

```
AdminTask.createGroup ('[-cn groupA -description a group of admins]')
```

- Using Jython list:

```
AdminTask.createGroup (['-cn', 'groupA', '-description', 'a group of admins'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGroup {-interactive}
```

- Using Jython string:

```
AdminTask.createGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createGroup (['-interactive'])
```

createUser

The **createUser** command creates a new user in the default repository or a repository that the parent command parameter specifies. This command creates a person entity and a login account entity in the virtual member manager.

Parameters and return values

-uid
Specifies the unique ID for the user that you want to create. Virtual member manager then creates a uniqueId value and a uniqueName value for the user. This parameter maps to the uid property in the virutal member manager. (String, required)

-password
Specifies the password for the user. This parameter maps to the password property in the virtual member manager. (String, required)

-confirmPassword
Specifies the password again to validate how it was entered for the password parameter. This parameter maps to the password property in virtual member manager. (String, optional)

-cn
Specifes the first name or given name of the user. This parameter maps to the cn property in virutal member manager. (String, optional)

-surname
Specifies the last name or family name of the user. This parameter maps to the sn property in virtual member manager. (String, optional)

-ibm-primaryEmail
Specifies the e-mail address of the user. This parameter maps to the ibm-PrimaryEmail property in the virtual member manager. (String, optional)

-parent
Specifies the repository in which you want to create the user. This parameter maps to the parent property in the virtual member manager. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createUser {-uid 123 -password tempPass -confirmPassword tempPass -cn Jane -sn
Doe -mail janedoe@ acme.com}
```

- Using Jython string:

```
AdminTask.createUser ('[-uid 123 -password tempPass -confirmPassword tempPass -cn Jane
-sn Doe -mail janedoe@ acme.com]')
```

- Using Jython list:

```
AdminTask.createUser (['-uid', '123', '-password', 'tempPass', '-confirmPassword', 'tempPass',
'-cn', 'Jane', '-sn', 'Doe', '-ibm -mail', 'janedoe@ acme.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createUser {-interactive}
```

- Using Jython string:

```
AdminTask.createUser ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createUser (['-interactive'])
```

deleteGroup

The **deleteGroup** command deletes a group in the virtual member manager. You cannot use this command to delete descendants. When this command completes, the group will be deleted from the repository.

Parameters and return values

-uniqueName
Specifies the unique name value for the group that you want to delete. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteGroup {-uniqueName cn=opera tors,cn=users,dc=yourco, dc=com}
  ```
- Using Jython string:
  ```
  AdminTask.deleteGroup ('[-uniqueName cn=ope rators,cn=users,dc=you rco,dc=com]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteGroup (['-uniqueName', 'cn =operators,cn=users,dc =yourco,dc=com'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteGroup {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.deleteGroup ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteGroup (['-interactive'])
  ```

deleteUser

The **deleteUser** command deletes a user from the virtual member manager. This includes a person object and an account object in the non-merged repositories.

Parameters and return values

-uniqueName
Specifies the unique name value for the user that you want to delete. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteUser {-uniqueName uid=dmey ers,cn=users,dc=yourco, dc=com}
  ```
- Using Jython string:
  ```
  AdminTask.deleteUser ('[-uniqueName uid= dmeyers,cn=users,dc= yourco,dc=com]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteUser ([' -uniqueName', 'uid=dm eyers,cn=users,dc=yourco, dc=com'])
  ```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteUser {-interactive}
```

- Using Jython string:

```
AdminTask.deleteUser ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteUser (['-interactive'])
```

duplicateMembershipOfGroup

Use the **duplicate Membership OfGroup** command to make a one group a member of all of the same groups as another group. For example, group A is in group B and group C. To add group D to the same groups as group A, use the **duplicate Membership OfGroup** command.

Parameters and return values

-copyToName
Specifies the name of the group to which you want to add the memberships of the group specified in the copyFromName parameter. (String, required)

-copyFromName
Specifies the name of the group from which you want to copy the group memberships for another group to use. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask duplicateMember shipOfGroup {-copyToName cn=operators,cn=groups, dc=yourco,dc=com
-copy FromName cn=admins,cn= groups,dc=yourco,dc=com}
```

- Using Jython string:

```
AdminTask.duplicateMembers hipOfGroup ('[-copyToName cn=operators,cn=groups, dc=yourco,dc=com
-copy FromName cn=admins,cn=gr oups,dc=yourco,dc=com]')
```

- Using Jython list:

```
AdminTask.duplicateMember shipOfGroup (['-copyToName', 'cn=operators,cn=groups, dc=yourco,dc=com',
'-copy FromName', 'cn=admins,cn =groups,dc=yourco,dc=com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask duplicateMember shipOfGroup {-interactive}
```

- Using Jython string:

```
AdminTask.duplicateMember shipOfGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.duplicateMember shipOfGroup (['-interactive'])
```

duplicateMembershipOfUser

Use the **duplicate Membership OfUser** command to make a one user a member of all of the same groups as another user. For example, user 1 is in group B and group C. To add user 2 to the same groups as user 1, use the **duplicate Membership OfUser** command.

Parameters and return values

-copyToName
Specifies the name of the user to which you want to add the memberships of the user specified in the copyFromName parameter. (String, required)

-copyFromName
Specifies the name of the user from which you want to copy the group memberships for another user to use. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask duplicateMember shipOfUser {-copyToName uid=meyersd,cn=users,dc =yourco,dc=com
-copy FromName uid=jhart,cn= users,dc=yourco,dc=com}
```

- Using Jython string:

```
AdminTask.duplicateMember shipOfUser ('[-copyToName uid=meyersd,cn=users,d c=yourco,dc=com
-copy FromName uid=jhart,cn =users,dc=yourco,dc=com]')
```

- Using Jython list:

```
AdminTask.duplicateMember shipOfUser (['-copyToName', 'uid=meyersd,cn=users, dc=yourco,dc=com',
'-copyFromName', 'uid= jhart,cn=users,dc=yourco, dc=com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask duplicateMember shipOfUser {-interactive}
```

- Using Jython string:

```
AdminTask.duplicateMember shipOfUser ('[-interactive]')
```

- Using Jython list:

```
AdminTask.duplicateMember shipOfUser (['-interactive'])
```

getGroup

The **getGroup** command retrieves the common name and description of a group.

Parameters and return values

-uniqueName
Specifies the unique name value for the group that you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getGroup {-uniqueName cn=operators, cn=groups,dc=yourco,dc=com}
```

- Using Jython string:

```
AdminTask.getGroup ('[ -uniqueName cn=operators, cn=groups,dc=yourco, dc=com]')
```

- Using Jython list:

```
AdminTask.getGroup ([' -uniqueName', 'cn=opera tors,cn=groups,dc=your co,dc=com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getGroup {-interactive}
```

- Using Jython string:

```
AdminTask.getGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getGroup (['-interactive'])
```

getMembershipOfGroup

The **getMembership OfGroup** command retrieves the groups of which a group is a member.

Parameters and return values

-uniqueName
Specifies the unique name value for the group whose group memberships you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getMebmership OfGroup {-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com}
  ```
- Using Jython string:
  ```
  AdminTask.getMebmership OfGroup ('[-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com]')
  ```
- Using Jython list:
  ```
  AdminTask.getMebmership OfGroup (['-uniqueName', 'uid=dmeyers,cn=users, dc=yourco,dc=com'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getMembership OfGroup {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getMembership OfGroup ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getMembership OfGroup (['-interactive'])
  ```

getMembershipOfUser

The **getMem bership OfUser** command retrieves the groups of which a user is a member.

Parameters and return values

-uniqueName
Specifies the unique name value for the user whose group memberships you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getMebmer shipOfUser {-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com}
  ```
- Using Jython string:
  ```
  AdminTask.getMebmership OfUser ('[-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com]')
  ```
- Using Jython list:
  ```
  AdminTask.getMebmership OfUser (['-uniqueName', 'uid=dmeyers,cn=users ,dc=yourco,dc=com'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getMembership OfUser {-interactive}
  ```
- Using Jython string:

```
AdminTask.getMembership OfUser ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getMembership OfUser (['-interactive'])
```

getMembersOfGroup

The **getMembers OfGroup** command retrieves the members of a group.

Parameters and return values

-uniqueName
Specifies the unique name value for the group whose members you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getMembersOf Group {-uniqueName cn=operators,cn=groups ,dc=yourco,dc=com}
```

- Using Jython string:

```
AdminTask.getMembersOf Group ['(-uniqueName cn=operators,cn=groups ,dc=yourco,dc=com]')
```

- Using Jython list:

```
AdminTask.getMembersOf Group [('-uniqueName', 'cn=operators,cn=groups ,dc=yourco,dc=com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getMembersOfGroup {-interactive}
```

- Using Jython string:

```
AdminTask.getMembersOfGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getMembersOfGroup (['-interactive'])
```

getUser

The **getUser** command retrieves information about a user in the virtual member manager.

Parameters and return values

-uniqueName
Specifies the unique name value for the user that you want to view. This parameter maps to the uniqueName property in the virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getUser {-user Name uid=dmeyers,cn=us ers,dc=yourco,dc=com}
```

- Using Jython string:

```
AdminTask.getUser ('[-use rName uid=dmeyers,cn= users,dc=yourco,dc=com]')
```

- Using Jython list:

```
AdminTask.getUser (['-use rName', 'uid=dmeyers,c n=users,dc=yourco,dc=com'])
```

Interactive mode example usage:

- Using Jacl:

  ```
  $AdminTask getUser {-interactive}
  ```

- Using Jython string:

  ```
  AdminTask.getUser ('[-interactive]')
  ```

- Using Jython list:

  ```
  AdminTask.getUser (['-interactive'])
  ```

removeMemberFromGroup

The **removeMember FromGroup** command removes a user or a group from a group.

Parameters and return values

-memberUniqueName
Specifies the unique name value for the user or group that you want to remove from the specified group. This parameter maps to the uniqueName property in virtual member manager. (String, required)

-groupUniqueName
Specifies the unique name value for the group from which you want to remove the user or group that you specified with the memberUniqueName paramter. This parameter maps to the uniqueName property in virtual member manager. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

  ```
  $AdminTask removeMember FromGroup {-memberUnique Name uid=meyersd,cn= users,dc=yourco,dc=com
  -groupUniqueName cn= admins,cn-groups,dc= yourco,dc=com}
  ```

- Using Jython string:

  ```
  AdminTask.removeMemberF romGroup ('[-memberUnique Name uid=meyersd,cn= users,dc=yourco,dc=com
  -groupUniqueName cn=a dmins,cn-groups,dc=your co,dc=com]')
  ```

- Using Jython list:

  ```
  AdminTask.removeMemberFrom Group (['-memberUniqueName', 'uid=meyersd,cn=users, dc=yourco,dc=com',
  '-groupUniqueName', 'cn=admins,cn-groups,dc= yourco,dc=com'])
  ```

Interactive mode example usage:

- Using Jacl:

  ```
  $AdminTask removeMember FromGroup {-interactive}
  ```

- Using Jython string:

  ```
  AdminTask.removeMemberFr omGroup ('[-interactive]')
  ```

- Using Jython list:

  ```
  AdminTask.removeMemberFr omGroup (['-interactive'])
  ```

searchGroups

Use the **searchGroups** command to find groups in the virtual member manager that match criteria that you provide. For example, you can use the **searchGroups** command to find all of the groups with a common name that begins with `IBM`. You can search for any virtual member manager property because the command is generic.

Parameters and return values

-cn
The first name or given name of the user. This parameter maps to the cn property in the virtual member manager. You must set this parameter or the description parameter, but not both. (String, optional)

-description
Specifies information about the group. This parameter maps to the description entity in a virtual member manager object. You must set this parameter or the cn parameter, but not both. (String, optional)

-timeLimit
Specifies the maximum amount of time in milliseconds that the search can run. The default value is no time limit. (String, optional)

-countLimit
Specifies the maximum number of results that you want returned from the search. By default, all groups found in the search are returned. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask searchGroups {cn *IBM*}
  ```
- Using Jython string:
  ```
  AdminTask.searchGroups ('[cn *IBM*]')
  ```
- Using Jython list:
  ```
  AdminTask.searchGroups (['cn', '*IBM*'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask searchGroups {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.searchGroups ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.searchGroups (['-interactive'])
  ```

searchUsers

Use the **searchUsers** command to find users in the virtual member manager that match criteria that you provide. For example, you can use the **searchUsers** command to find all of the telephone numbers that contain `919`. You can search for any virtual member manager property because the command is generic.

Parameters and return values

-principalName
Specifies the principal name oft he user that is used as the logon ID for the user in the system. This parameter maps to the principalName property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)

-uid
Specifies the unique ID value for the user for whom you want to search. This parameter maps to the uid property in virtual member manage. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)

-cn
Specifies the first name or given name of the user. This parameter maps to the cn property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)

-sn
Specifies the last name or family name of the user. This parameter maps to the sn property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)

-ibm-primaryEmail
Specifies the email address of the user. This parameter maps to the ibm-PrimaryEmail property in the virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)

-timeLimit
Specifies the maximum amount of time in milliseconds that the search can run. The default is not time limit. (String, optional)

-countLimit
Specifies the maximum number of results that you want returned from the search. By default, all users found int he search are returned. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask searchUsers {-principalName */IBM/US*}
```

- Using Jython string:

```
AdminTask.searchUsers (' [-principalName */IBM/US*]')
```

- Using Jython list:

```
AdminTask.searchUsers (['-principalName', '*/IBM/US*'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask searchUsers {-interactive}
```

- Using Jython string:

```
AdminTask.searchUsers ('[-interactive]')
```

- Using Jython list:

```
AdminTask.searchUsers (['-interactive'])
```

updateGroup

The **updateGroup** command updates the common name or the description of a group.

Parameters and return values

-uniqueName
Specifies the unique name value for the group for which you want to modify the properties. This parameter maps to the uniqueName property in virtual member manager. (String, required)

-cn
Specifies the new common name used for the group. This parameter maps to the cn property in virtual member manager. (String, optional)

-description
Specifies the new information about the group. This parameter maps to the description entity in a virtual member manager object. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateGroup {-uniqueName cn=opera tors,cn=groups,dc=yourco ,dc=com -cn groupA}
```

- Using Jython string:

```
AdminTask.updateGroup ('[-uniqueName cn=oper ators,cn=groups,dc=your co,dc=com -cn groupA]')
```

- Using Jython list:

```
AdminTask.updateGroup ([' -uniqueName', 'cn=oper ators,cn=groups,dc=yourco, dc=com', '-cn', 'groupA'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateGroup {-interactive}
```

- Using Jython string:

```
AdminTask.updateGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateGroup (['-interactive'])
```

updateUser

The **updateUser** command updates the following properties: uniqueName, uid, password, cn, sn, or ibm-primaryEmail.

Parameters and return values

-uniqueName
Specifies the unique name value for the user for which you want to modify the properties. This parameter maps to the uniqueName property in virtual member manager. (String, required)

-uid
Specifies the new unique ID value for the user. This parameter maps to the uid property in virtual member manager. (String, optional)

-password
Specifies the new password for the user. This parameter maps to the password property in virtual member manager. (String, optional)

-confirmPassword
Specifies the password again to validate how it was entered on the password parameter. This parameter maps to the password property in virtual member manager. (String, optional)

-cn
Specifies the new first name or given name of the user. This parameter maps to the cn property in virtual member manager. (String, optional)

-surname
Specifies the new last name or family name of the user. This parameter maps to the sn property in virtual member manager. (String, optional)

-ibm-primaryEmail
Specifies the new e-mail address of the user. This parameter maps to the mail property in virtual member manager. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateUser {-uniqueName uid=dmey ers,cn=users,dc=yourco, dc=com -uid 123}
```

- Using Jython string:

```
AdminTask.updateUser ('[-uniqueName uid= dmeyers,cn=users,dc= yourco,dc=com -uid 123]')
```

- Using Jython list:

```
AdminTask.updateUser (['-uniqueName', 'uid =dmeyers,cn=users,dc= yourco,dc=com', '-uid', '123'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateUser {-interactive}
```

- Using Jython string:

```
AdminTask.updateUser ('[-interactive]')
```

- Using Jython list:

```
AdminTask.updateUser (['-interactive'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

DescriptivePropCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the DescriptivePropCommands group can be used to create, delete, and manage key manager setting in your configuration.

The DescriptivePropCommands command group for the AdminTask object includes the following commands:

- "deleteDescriptiveProp"
- "getDescriptiveProp" on page 903
- "listDescriptiveProp" on page 903
- "modifyDescriptiveProp" on page 904

deleteDescriptiveProp

The **deleteDescriptiveProp** command deletes key manager settings from the configuration.

Target object

None

Parameters and return values

-parentDataType
(String, required)

-parentClassName
(String, required)

-parentScopeName
(String, optional)

-name
(String, required)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteDescrip tiveProp {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.deleteDescrip tiveProp ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteDescrip tiveProp (['-interactive'])
  ```

getDescriptiveProp

The **getDescriptiveProp** command obtains information about key manager settings.

Target object

None

Parameters and return values

-parentDataType
(String, required)

-parentClassName
(String, required)

-parentScopeName
(String, optional)

-name
(String, required)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getDescripti veProp {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getDescriptiv eProp ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getDescriptive Prop (['-interactive'])
  ```

listDescriptiveProp

The **listDescriptiveProp** command lists the key managers within a particular management scope.

Target object

None

Parameters and return values

-parentDataType
(String, required)

-parentClassName
(String, required)

-parentScopeName
(String, optional)

-displayObjectName
Set the value of this parameter to `true` to list the key manager objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key manager name and management scope. (Boolean, optional)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listDescrip tiveProp {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listDescrip tiveProp ('[-interact ive]')
  ```
- Using Jython list:
  ```
  AdminTask.listDescrip tiveProp (['-interact ive'])
  ```

modifyDescriptiveProp

The **modifyDescriptiveProp** command modifies the settings of an existing key manager.

Target object

None

Parameters and return values

-parentDataType
(String, required)

-parentClassName
(String, required)

-parentScopeName
(String, optional)

-name
(String, required)

-value
(String, optional)

-type
(String, optional)

-displayNameKey
(String, optional)

-nlsRangeKey
(String, optional)

-hoverHelpKey
(String, optional)

-range
(String, optional)

-inclusive
(Boolean, optional)

-firstClass
(Boolean, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyDescrip tiveProp {-interactive}
```

- Using Jython string:

```
AdminTask.modifyDescrip tiveProp ('[-interacti ve]')
```

- Using Jython list:

```
AdminTask.modifyDescrip tiveProp (['-interactiv e'])
```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

ManagementScopeCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. Inbound and outbound management scopes represent opposing directions during the connection handshake process. The commands and parameters in the ManagementScopeCommands group can be used to create, delete, and list management scopes.

The ManagementScopeCommands command group for the AdminTask object includes the following commands:

- "deleteManagementScope"
- "getManagementScope" on page 906
- "listManagementScopes" on page 906

deleteManagementScope

The **deleteManage mentScope** command deletes a management object from the configuration.

Target object

None

Parameters and return values

- scopeName
The name that uniquely identifies the management scope. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

getManagementScope

The **getManagementScope** command displays the setting of a management scope object.

Target object

None

Parameters and return values

- scopeName
The name that uniquely identifies the management scope. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

listManagementScopes

The **listManagementScopes** command lists the management scopes in the configuration.

Target object

None

Parameters and return values

- displayObjectName
Set the value to true to display the object names of the management scope. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

AuthorizationGroupCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the AuthorizationGroupCommands group can be used to create and manage authorization groups.

The AuthorizationGroupCommands command group for the AdminTask object includes the following commands:

- “addResourceToAuthorizationGroup”
- “createAuthorizationGroup” on page 908
- “deleteAuthorizationGroup” on page 909
- “listAuthorizationGroups” on page 909
- “listAuthorizationGroupsForGroupID” on page 910
- “listAuthorizationGroupsForUserID” on page 911
- “listAuthorizationGroupsOfResource” on page 911
- “listResourcesOfAuthorizationGroup” on page 912
- “listResourcesForGroupID” on page 913
- “listResourcesForUserID” on page 913
- “mapGroupsToAdminRole” on page 914
- “mapUsersToAdminRole” on page 915
- “removeGroupsFromAdminRole” on page 916
- “removeResourceFromAuthorizationGroup” on page 917
- “removeUsersFromAdminRole” on page 918

addResourceToAuthorizationGroup

The **addResourceToAuthorizationGroup** command adds a resource instance to an existing authorization group. A resource instance cannot belong to more than one authorization group.

Target object

None

Parameters and return values

- authorizationGroupName
The name of the authorization group. (String, required)

- resourceName
The name of the resource instance that you want to add to an authorization group. (String, required)

The resourceName parameter should be in the following format:

```
ResourceType=ResourceName
```

where ResourceType is one of the following values: Application, Server, ServerCluster, Node, NodeGroup

ResourceName is the name of the resource instance, for example, server1.

The following are example uses of the resourceName parameter:

- Node=node1:Server=server1

  This example uniquely identifies server1. node1 is required if another server1 exists on a different node.
- Application=app1

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask addResourceToAuthorizationGroup {-authorizationGroupName groupName -resourceName Application=app1}
  ```
- Using Jython string:
  ```
  AdminTask.addResourceToAuthorizationGroup('[-authorizationGroupName groupName -resourceName Application=app1]')
  ```
- Using Jython list:
  ```
  AdminTask.addResourceToAuthorizationGroup(['-authorizationGroupName', 'groupName', '-resourceName', 'Application=app1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask addResourceToAuthorizationGroup {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.addResourceToAuthorizationGroup ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.addResourceToAuthorizationGroup (['-interactive'])
  ```

createAuthorizationGroup

The **createAuthorizationGroup** command creates a new authoirzation group. When you create a new authorization group, no members are associated with it. Also, no user to administrative role mapping for the authorization table is associated with the authorization group.

Target object

None

Parameters and return values

- authorization GroupName
The name of the authorization group that you want to create. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask createAuthorizationGroup {-authorizationGroupName groupName}
  ```
- Using Jython string:
  ```
  AdminTask.createAuthorizationGroup('[-authorizationGroupName groupName]')
  ```
- Using Jython list:
  ```
  AdminTask.createAuthorizationGroup(['-authorizationGroupName', 'groupName'])
  ```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createAuthorizationGroup -interactive
```

- Using Jython string:

```
AdminTask.createAuthorizationGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createAuthorizationGroup (['-interactive'])
```

deleteAuthorizationGroup

The **deleteAuthorizationGroup** command deletes an existing authorization group. When you delete an authorization group, the authorization table that corresponds is also deleted.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group that you want to delete. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteAuthorizationGroup {-authorizationGroupName groupName}
```

- Using Jython string:

```
AdminTask.deleteAuthorizationGroup('[-authorizationGroupName groupName]')
```

- Using Jython list:

```
AdminTask.deleteAuthorizationGroup(['-authorizationGroupName', 'groupName'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteAuthorizationGroup  {-interactive}
```

- Using Jython string:

```
AdminTask.deleteAuthorizationGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteAuthorizationGroup (['-interactive'])
```

listAuthorizationGroups

The **listAuthorizationGroupslistAuthoriz ationGroups** command lists the existing authorization groups.

Target object

None

Parameters and return values

- Parameters: None
- Returns: A list of short names of all existing authorization groups. (String [])

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroups
  ```
- Using Jython:
  ```
  AdminTask.listAuthorizationGroups()
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroups {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroups ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroups (['-interactive'])
  ```

listAuthorizationGroupsForGroupID

The **listAuthorizationGroupsForGroupID** command lists all of the authorization groups to which a given user group has access. This command lists the authorization groups and the granted roles for each authorization group. The group ID can be a short name or a fully qualified domain name if the LDAP user registry is being used. This command will list `cell` as a group if the user has cell level access.

Target object

None

Parameters and return values

- groupid
The ID of the user group. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsForGroupID {-groupid userGroupName}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsForGroupID('[-groupid userGroupName]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsForGroupID(['-groupid', 'userGroupName'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsForGroupID {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsForGroupID ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsForGroupID (['-interactive'])
  ```

listAuthorizationGroupsForUserID

The **listAuthorizationGroupsForUserID** command lists all of the authorization groups to which a given user has access. This command lists the authorization groups and the granted roles for each authorization group. The user ID and the group ID can be a short name or a fully qualified domain name if the LDAP user registry is being used. This command will list `cell` as a group if the user has cell level access.

Target object

None

Parameters and return values

- userid
The ID of the user. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsForUserID{-userid userName}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsForUserID('[-userid userName]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsForUserID(['-userid', 'userName'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsForUserID {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsForUserID ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsForUserID (['-interactive'])
  ```

listAuthorizationGroupsOfResource

The **listAuthorizationGroupsOfResource** command lists authorization groups for a given resource. If the value of the traverseContainedObjects parameter is false, only the authorization group of the resource is returned. If the value of the traverseContainedObjects parameter is true, it returns the authorization group of the resource and the authorization groups of all the parent resources in the containment tree.

Target object

None

Parameters and return values

- resourceName
The name of the resource. (String, required)

The resourceName parameter must be in the following format:

```
ResourceType=ResourceName
```

where `ResourceType` can be any one of the following values: Application, Server, ServerCluster, Node, or NodeGroup.

`ResourceName` is the name of the resource instance, for example, server1.

The following are examples of the resourceName parameter:

```
Node=node1:Server=server
```

This example uniquely identifies *server1*. The name of the node is required if a server on a different node uses the same server name.

```
Application=app1
```

- traverseContained Resources
Finds the authorization groups of all the parent resources by traversing the resource containment tree upwards. The default value is false. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsOfResource {-resourceName Application=app1}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsOfResource('[-resourceName Application=app1]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsOfResource(['-resourceName', 'Application=app1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listAuthorizationGroupsOfResource {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listAuthorizationGroupsOfResource ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listAuthorizationGroupsOfResource (['-interactive'])
  ```

listResourcesOfAuthorizationGroup

The **listResourcesOfAuthorizationGroup** command lists all of the resources within the given authorization group.

Target object

None

Parameters and return values

- authorizationGroupName
The name of the authorization group. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listResourcesOfAuthorizationGroup {-authorizationGroupName groupName}
  ```
- Using Jython string:
  ```
  AdminTask.listResourcesOfAuthorizationGroup('[-authorizationGroupName groupName]')
  ```
- Using Jython list:

```
AdminTask.listResourcesOfAuthorizationGroup(['-authorizationGroupName', 'groupName'])
```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listResourcesOfAuthorizationGroup {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listResourcesOfAuthorizationGroup ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listResourcesOfAuthorizationGroup (['-interactive'])
  ```

listResourcesForGroupID

The **listResourcesForGroupID** command lists all the objects that a given group has access to. This command lists the resources and the granted roles for each resource. The resources that this command returns include the resources from the authorization groups to which the user group is granted roles and the resources that are descendants of the resources with in authorization groups to which the user group is granted access to any role. The group ID can be a short name or fully qualified domain name if a LDAP user registry is used.

Target object

None

Parameters and return values

- groupid
The ID of the user group. (String, required)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listResourcesForGroupID {-groupid userGroupName}
  ```
- Using Jython string:
  ```
  AdminTask.listResourcesForGroupID('[-groupid userGroupName]')
  ```
- Using Jython list:
  ```
  AdminTask.listResourcesForGroupID(['-groupid', 'userGroupName'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listResourcesForGroupID {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listResourcesForGroupID ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listResourcesForGroupID (['-interactive'])
  ```

listResourcesForUserID

The **listResourcesForUserID** command lists all the objects that a given user has access to. This command lists the resources and the granted roles for each resource. The resources that this command returns include the resources from the authorization groups to which the user is granted roles and the

resources that are descendants of the resources with in authorization groups to which the user is granted access to any role. The user ID can be a short name or fully qualified domain name if a LDAP user registry is used.

Target object

None

Parameters and return values

- userid
The ID of the user. (String, required).

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask listResourcesForUserID {-userid userName }
```
- Using Jython string:
```
AdminTask.listResourcesForUserID('[-userid userName]')
```
- Using Jython list:
```
AdminTask.listResourcesForUserID(['-userid', 'userName'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask listResourcesForUserID {-interactive}
```
- Using Jython string:
```
AdminTask.listResourcesForUserID ('[-interactive]')
```
- Using Jython list:
```
AdminTask.listResourcesForUserID (['-interactive'])
```

Example output:

```
{deployer=[], operator=[],  administrator=[cells/IBM-LP1 6L31HVE8Cell07/clusters/C1| cluster.xml,
 cells/IBM-LP16L 31HVE8Cell07/nodes/IBM-LP16L 31HVE8Node05/servers/cm1|ser ver.xml],
 monitor=[], configurator=[]}
```

mapGroupsToAdminRole

The **mapGroupsToAdminRole** command maps group IDs to one or more administrative roles in an authorization group. The name of the authorization group that you provide determines which authorization table will be used. If you do not specify an authorization group name, the mapping is done to the cell level authorization table. The group ID can be a short name or a fully qualified domain name if the LDAP user registry is used.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group. If you do not specify this parameters, the cell level authorization group is assumed. (String, optional)

- roleName
The name of the administrative role. (String, required)

- groupids
The list of group IDs that will mapped to the administrative role. (String[], required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask mapGroupsToAdminRole {-authorizationGroupName groupName - roleName administrator -groupids group1}
```

- Using Jython string:

```
AdminTask.mapGroupsToAdminRole('[-authorizationGroupName groupName -roleName administrator -groupids group1]')
```

- Using Jython list:

```
AdminTask.mapGroupsToAdminRole(['-authorizationGroupName', 'groupName', '-roleName', 'administrator',
'-groupids', 'group1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask mapGroupsToAdminRole {-interactive}
```

- Using Jython string:

```
AdminTask.mapGroupsToAdminRole ('[-interactive]')
```

- Using Jython list:

```
AdminTask.mapGroupsToAdminRole (['-interactive'])
```

mapUsersToAdminRole

The **mapUsersToAdminRole** command maps user IDs to one or more administrative roles in the authorization group. The name of the authorization group that you provide determines the authorization table. If you do not specify the name of the authorization group, the mapping is done to the cell level authorization table. The user ID can be a short name or fully qualified domain name in case LDAP user registry is used.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional)

- roleName
The name of the administrative role. (String, required)

- userids
The list of user IDs that will be mapped to the administrative role (String[], required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask mapUsersToAdminRole {-authorizationGroupName groupName - roleName administrator -userids user1}
```

- Using Jython string:

```
AdminTask.mapUsersToAdminRole('[-authorizationGroupName groupName -roleName administrator -userids user1]')
```

- Using Jython list:

```
AdminTask.mapUsersToAdminRole(['-authorizationGroupName', 'groupName', '-roleName', 'administrator',
'-userids', 'user1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask mapUsersToAdminRole {-interactive}
```

- Using Jython string:

```
AdminTask.mapUsersToAdminRole ('[-interactive]')
```

- Using Jython list:

```
AdminTask.mapUsersToAdminRole (['-interactive'])
```

removeGroupsFromAdminRole

The **removeGroupsFromAdminRole** command removes previously mapped group IDs from administrative roles in the authorization group. The name of the authorization group that you provide determines which authorization table is involved. If you do not specify an authorization group name, the group IDs are removed from the cell level authorization table. The group ID can be a short name or fully qualified domain name if a LDAP user registry is used.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional)

- roleName
The name of the administrative role. (String, required)

- userids
A list of group IDs that you want to remove from the administrative role. (String[], required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeGroupsFromAdminRole {-authorizationGroupName groupName - roleName administrator -groupids group1}
```

- Using Jython string:

```
AdminTask.removeGroupsFromAdminRole('[-authorizationGroupName groupName -roleName administrator -groupids group1]')
```

- Using Jython list:

```
AdminTask.removeGroupsFromAdminRole(['-authorizationGroupName', 'groupName', '-roleName', 'administrator',
 '-groupids', 'group1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeGroupsFromAdminRole {-interactive}
```

- Using Jython string:

```
AdminTask.removeGroupsFromAdminRole ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeGroupsFromAdminRole (['-interactive'])
```

removeResourceFromAuthorizationGroup

The **removeResourceFromAuthorizationGroup** command removes resources from an existing authorization group. If you do not specify the authorization group, it will be determined and the resource will be removed from that authorization group.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group. (String, optional)

- resourceName
The name of the resource instance that you want to remove from the authorization group. (String, required)

The resourceName parameter must be in the following format:

```
ResourceType=ResourceName
```

where the `ResourceType` can be any of the following: Application, Server, ServerCluster, Node, or NodeGroup.

The `ResourceName` is the name of the resource instance, for example, server1.

The following are examples of the resourceName parameter:

```
Node=node1:Server=server1
```

This example uniquely identifies server1. node1 is required if the name of the server exists on multiple nodes.

```
Application=app1
```

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeResourceFromAuthorizationGroup {-authorizationGroupName groupName -resourceName Application=app1}
```

- Using Jython string:

```
AdminTask.removeResourceFromAuthorizationGroup('[-authorizationGroupName groupName -resourceName Application=app1]')
```

- Using Jython list:

```
AdminTask.removeResourceFromAuthorizationGroup(['-authorizationGroupName',  'groupName', '-resourceName',
 'Application=app1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeResourceFromAuthorizationGroup {-interactive}
```

- Using Jython string:

```
AdminTask.removeResourceFromAuthorizationGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeResourceFromAuthorizationGroup (['-interactive'])
```

removeUsersFromAdminRole

The **removeUsersFromAdminRole** command removes previously mapped user IDs from administrative roles in the authorization group. The name of the authorization group that you provide determines which authorization table is involved. If you do not specify an authorization group name, the user ID from the cell level authorization table will be used. The user ID can be a short name or a fully qualified domain name if a LDAP user registry is used.

Target object

None

Parameters and return values

- authorizationGroup Name
The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional)

- roleName
The name of the administrative role. (String, required)

- userids
A list of user IDs that you want to remove from the administrative role. (String[], required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeUsersFromAdminRole {-authorizationGroupName groupName - roleName administrator -userids user1}
```

- Using Jython string:

```
AdminTask.removeUsersFromAdminRole('[-authorizationGroupName groupName -roleName administrator -userids user1]')
```

- Using Jython list:

```
AdminTask.removeUsersFromAdminRole(['-authorizationGroupName', 'groupName', '-roleName', 'administrator',
'-userids', 'user1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeUsersFromAdminRole {-interactive}
```

- Using Jython string:

```
AdminTask.removeUsersFromAdminRole ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeUsersFromAdminRole (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

ChannelFrameworkManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security. The commands and parameters in the ChannelFrameworkManagement group can be used to create and manage transport channels and transport channel chains.

The ChannelFrameworkManagement command group for the AdminTask object includes the following commands:

- “createChain”
- “deleteChain” on page 920
- “listChainTemplates” on page 921
- “listChains” on page 921

createChain

The **createChain** command creates a new chain of transport channels that are based on a chain template.

Target object

The instance of the transport channel service under which the new chain is created. (ObjectName, required)

Required parameters and return values

- template
The chain template on which to base the new chain. (ObjectName, required)

- name
The name of the new chain. (String, required)

- endPoint
The name of the end point to be used by the instance of the TCP inbound channel in the new chain if the chain is an inbound chain. (ObjectName, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createChain (cells/ rohitbuildCell01/nodes/rohit buildCellManager01/servers/
 dmgr|server.xml#TransportCha  nnelService_1) {-template  WebContainer(templates/ chains|webcontainer-chains.xml #Chain_1)
 -name trialChain1}

$AdminTask createChain (cells/ rohitbuildCell01/nodes/rohitb uildCellManager01/servers/dmgr
 |server.xml#TransportChannel  Service_1) {-template  WebContainer(templates/ chains|webcontainer-chains.xml# Chain_1)
 -name trialChain1   -endPoint (cells/rohitbuild Cell01/nodes/rohitbuildCellMa nager01|serverindex.xml#End Point_3) }
```

- Using Jython string:

```
  AdminTask.createChain('cells/ rohitbuildCell01/nodes/rohitbu ildCellManager01/servers/dmgr|
 server.xml#TransportChannelSer  vice_1', '[-template "WebConta iner(templates/chains|webconta iner-chains.xml#Chain_1)" -name
 trialChain]')
```

```
AdminTask.createChain('cells/ rohitbuildCell01/nodes/rohit buildCellManager01/servers/dmgr
 |server.xml#TransportChannel Service_1',  '[-template "WebCo ntainer(templates/chains|webc ontainer-chains.xml#Chain_1)"  -name
 trialChain -endPoint   "(cells/rohitbuildCell01/nodes/ rohitbuildCellManager01|server index.xml#EndPoint_3)"]')
```

- Using Jython list:

```
AdminTask.createChain('cells/ rohitbuildCell01/nodes/ rohitbuildCellManager01/serve
 rs/dmgr|server.xml# TransportChannelService_1',   ['-template', "WebContainer (templates/chains|webcontaine
 r-chains.xml#Chain_1)",  '-name', 'trialChain'])
```

```
AdminTask.createChain('cells/ rohitbuildCell01/nodes/rohit buildCellManager01/servers/
 dmgr|server.xml#TransportCha nnelService_1',  ['-template',  "WebContainer(templates/chains |webcontainer-chains.xml#Cha in_1)",
 '-name', 'trialChain',  '-endPoint',  "(cells/rohitbu ildCell01/nodes/rohitbuildCe llManager01|serverindex.xml#
 EndPoint_3)"])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createChain {-interactive}
```

- Using Jython string:

```
AdminTask.createChain ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createChain (['-interactive'])
```

deleteChain

The **deleteChain** command deletes an existing chain and, optionally, the transport channels in the chain.

Target object

The chain to be deleted. (Object name, required)

Required parameters and return values

- deleteChannels
If the value of this attribute is true, non-shared transport channels used by the specified chain will be deleted. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteChain trialChain1 (cells/rohitbuildCell01/nodes/roh
 itbuildCellManager01/servers/dmgr | server.xml#Chain_1093554462922)
```

```
$AdminTask deleteChain trialChain (cells/rohitbuildCell01/nodes/roh
 itbuildCellManager01/servers/dmgr  |server.xml#Chain_1093554378078)  {-deleteChannels true}
```

- Using Jython string:

```
  AdminTask.deleteChain('trialChain 1(cells/rohitbuildCell01/nodes/roh
 itbuildCellManager01/servers/dmgr|  server.xml#TransportChannelSer vice_1)')
```

```
AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/rohi
 tbuildCellManager01/servers/dmgr|  server.xml#TransportChannelService _1)', '[-deleteChannels true]')
```

- Using Jython list:

```
AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/roh
 itbuildCellManager01/servers/dmgr|  server.xml#TransportChannelService _1)')
```

```
AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/rohi
 tbuildCellManager01/servers/dmgr|  server.xml#TransportChannelServic e_1)', ['-deleteChannels', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteChain {-interactive}
```

- Using Jython string:

```
AdminTask.deleteChain ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteChain (['-interactive'])
```

listChainTemplates

The **listChain Templates** command displays a list of templates that you can use to create chains in this configuration. All templates have a certain type of transport channel as the last transport channel in the chain.

Target object

None

Required parameters and return values

- acceptorFilter
The templates returned by this method all have a transport channel instance of the specified type as the last transport channel in the chain. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listChainTemplates {}
$AdminTask listChainTemplates  "-acceptorFilter WebContainer InboundChannel"
```

- Using Jython string:

```
AdminTask.listChainTemplates()
AdminTask.listChainTemplates ('[-acceptorFilter WebCont ainerInboundChannel]')
```

- Using Jython list:

```
AdminTask.listChainTemplates()
AdminTask.listChainTemplates (['-acceptorFilter', 'WebC ontainerInboundChannel'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listChainTemplates {-interactive}
```

- Using Jython string:

```
AdminTask.listChainTemplates ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listChainTemplates (['-interactive'])
```

listChains

The **listChains** command lists all the chains that are configured under a particular instance of the transport channel service.

Target object

The instance of the transport channel service under which the chains are configured. (ObjectName, required)

Required parameters and return values

- acceptorFilter
The chains that are returned by this parameter will have a transport channel instance of the type that you specify as the last transport channel in the chain. (String, optional)

- endPointFilter:

The chains returned by this parameter will have a TCP inbound channel using an end point with the name that you specify.(String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listChains (cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)
$AdminTask listChains (cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)  {-acceptorFilter WebConta inerInboundChannel}
$AdminTask listChains (cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328) {-end PointFilter WC_adminhost}
```

- Using Jython string:

```
AdminTask.listChains('(cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)')
AdminTask.listChains('(cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)',  '[-acceptorFilter WebContai nerInboundChannel]')
AdminTask.listChains('(cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)',  '[-endPointFilter WC_adminhost]')
```

- Using Jython list:

```
AdminTask.listChains('(cells/ rohitbuildCell01/nodes/rohit buildNode01/servers/server2|
 server.xml#TransportChannel Service_1093445762328)')
AdminTask.listChains('(cells /rohitbuildCell01/nodes/roh itbuildNode01/servers/server 2
 |server.xml#TransportChanne lService_1093445762328)',  ['-acceptorFilter', 'WebCon tainerInboundChannel'])
AdminTask.listChains('(cells /rohitbuildCell01/nodes/roh itbuildNode01/servers/server
 2|server.xml #TransportChanne lService_1093445762328)',  ['-endPointFilter', 'WC_admi nhost'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listChains {-interactive}
```

- Using Jython string:

```
AdminTask.listChains ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listChains (['-interactive'])
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

SpnegoTAICommands group for the AdminTask object (deprecated)

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the SpnegoTAICommands group can be used to create and manage configurations that are used by the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI).

Note:

In WebSphere Application Server Version 6.1, a trust association interceptor (TAI) that uses the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) to securely negotiate and authenticate HTTP requests for secured resources was introduced. In WebSphere Application

Server 7.0, this function is now deprecated. SPNEGO Web authentication has taken its place to provide dynamic reload of the SPNEGO filters and to enable fallback to the application login method.

The SpnegoTAICommands command group for the AdminTask object includes the following commands:

- "addSpnegoTAIProperties"
- "deleteSpnegoTAIProperties" on page 924
- "modifySpnegoTAIProperties" on page 925
- "showSpnegoTAIProperties" on page 926
- "createKrbConfigFile" on page 926

addSpnegoTAIProperties

The **addSpnego TAI Properties** command adds properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for the application server.

Target object

None

Parameters and return values

-spnId
This is the SPN identifier for the group of custom properties that are to be defined with this command. If you do not specify this parameter, an unused SPN identifier is assigned. (String, optional)

-host
Specifies the host name portion in the SPN used by the SPNEGO TAI to establish a Kerberos secure context. (String, required)

-filter
Defines the filtering criteria used by the class specified with the above attribute. If no filter is specified, all HTTP requests are subject to SPNEGO authentication. (String, optional)

-filterClass
Specifies the name of the Java class used by the SPNEGO TAI to select which HTTP requests will be subject to SPNEGO authentication. If no filter class is specified, the default filter class, com.ibm.ws.security.spnego.HTTPHeaderFilter, is used. (String, optional)

-noSpnegoPage
Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application if it does not support SPNEGO authentication. (String, optional).

If you do not specify the noSpnegoPage attribute then the default is used:

```
"<html><head><title> SPNEGO authentication is not supported. </title></head>" +
"<body>SPNEGO authe ntication is not supported on this client.</body> </html>";
```

-ntlmTokenPage
Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application when the SPNEGO token received by the interceptor after the challenge-response handshake contains a NT LAN manager (NTLM) token instead of the expected SPNEGO token. (String, optional).

If you do not specify the ntlmTokenPage attribute then the default is used:

```
"<html><head><title> An NTLM Token was received.</title> </head>" + "<body>Your bro wser configuration
is correct, but you have not logged into a supported Windows Domain." + "<p>Please login to the application
using the normal login page.</html>";
```

-trimUserName
Specifies whether (`true`) or not (`false`) the SPNEGO TAI is to remove the suffix of the principal user name, starting from the `@` that precedes the Kerberos realm name. If this attribute is set to true, the suffix of the principal user name is removed. If this attribute is set to `false`, the suffix of the principal name is retained. The default value used is `true`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask addSpnegoTAIProp erties -host myhost.ibm.com -filter user-agent%=IE 6
  ```
- Using Jython string:
  ```
  AdminTask.addSpnegoTAIProp erties ('[-host myhost.ibm. com -filter user-agent%=IE 6]')
  ```
- Using Jython list:
  ```
  AdminTask.addSpnegoTAIProp erties (['-host', 'myhost. ibm.com', '-filter', 'user -agent%=IE', '6'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask addSpnegoTAI Properties -interactive
  ```
- Using Jython string:
  ```
  AdminTask.addSpnegoTAIPro perties ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.addSpnegoTAIPro perties ['-interactive'])
  ```

deleteSpnegoTAIProperties

The **deleteSpnego TAIProperties** command deletes properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.

Target object

None

Parameters and return values

-spnId
The SPN identifier for the group of custom properties that are to be deleted with this command. If you do not specify this parameter, all SPNEGO TAI custom properties are deleted. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask deleteSpnegoT AIProperties {-spnId 2}
  ```
- Using Jython string:
  ```
  AdminTask.deleteSpnegoT AIProperties ('[-spnId 2]')
  ```
- Using Jython list:
  ```
  AdminTask.deleteSpnegoTAI Properties (['-spnId', '2'])
  ```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteSpnegoTAI Properties -interactive
```

- Using Jython string:

```
AdminTask.deleteSpnegoTAI Properties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteSpnegoTAI Properties ['-interactive'])
```

modifySpnegoTAIProperties

The **modifySpnego TAIProperties** command modifies the properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.

Target object

None

Parameters and return values

-spnId
The SPN identifier for the group of custom properties that are to be defined with this command. (String, required)

-host
Specifies the host name portion in the SPN used by the SPNEGO TAI to establish a Kerberos secure context. (String, optional)

-filter
Defines the filtering criteria used by the class specified with the above attribute. (String, optional)

-filterClass
Specifies the name of the Java class used by the SPNEGO TAI to select which HTTP requests will be subject to SPNEGO authentication. If no class is specified, all HTTP requests will be subject to SPNEGO authentication. (String, optional)

-noSpnegoPage
Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application if it does not support SPNEGO authentication. (String, optional)

-ntlmTokenPage
Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application when the SPNEGO token received by the interceptor after the challenge-response handshake contains a NT LAN manager (NTLM) token instead of the expected SPNEGO token. (String, optional)

-trimUserName
Specifies whether (`true`) or not (`false`) the SPNEGO TAI is to remove the suffix of the principal user name, starting from the `"@"` that precedes the Kerberos realm name. If this attribute is set to true, the suffix of the principal user name is removed. If this attribute is set to `false`, the suffix of the principal name is retained. The default value used is `true`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifySpnegoTAI PROPERTIES -spnId 1 -filter host==myhost.company.com
```

- Using Jython string:

```
AdminTask.modifySpnegoTAI PROPERTIES ('[-spnId 1 -filter host==myhost.com pany.com]')
```

- Using Jython list:

```
AdminTask.modifySpnegoTAI PROPERTIES (['-spnId', '1', '-filter', 'host==my host.company.com'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifySpnegoTAI Properties -interactive
```

- Using Jython string:

```
AdminTask.modifySpnegoTAI Properties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifySpnegoTAI Properties ['-interactive'])
```

showSpnegoTAIProperties

The **showSpnego TAI Properties** command displays the properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.

Target object

None

Parameters and return values

-spnId
The service principal name (SPN) identifier for the group of custom properties that are to be displayed with this command. If you do not specify this parameter, all SPNEGO TAI custom properties are displayed. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showSpnegoTAI Properties -spnId 1
```

- Using Jython string:

```
AdminTask.showSpnegoTAI Properties ('[-spnId 1]')
```

- Using Jython list:

```
AdminTask.showSpnegoTAI Properties (['-spnId', '1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showSpnegoTAI Properties -interactive
```

- Using Jython string:

```
AdminTask.showSpnegoTAI Properties ('[-interact ive]')
```

- Using Jython list:

```
AdminTask.showSpnegoTAI Properties ['-interact ive'])
```

createKrbConfigFile

The **createKrb ConfigFile** command creates the Kerberos configuration file for use with the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.

Target object

None

Parameters and return values

-krbPath
Provides the fully qualified file system location of the Kerberos configuration (krb5.ini or krb5.conf) file. (String, required)

-realm
Provides the Kerberos realm name. The value of this attribute is used by the SPNEGO TAI to form the Kerberos service principal name for each of the hosts specified with the property com.ibm.ws.security.spnego.SPN<id>.hostname (String, required)

-kdcHost
Provides the host name of the Kerberos Key Distribution Center (KDC). (String, required)

-kdcPort
Provides the port number of the KDC. The default value, if not specified, is 88. (String, optional)

-dns
Provides the default domain name service (DNS) that is used to produce a fully qualified host name. (String, required)

-keytabPath
Provides the file system location of the Kerberos keytab file. (String, required)

-encryption
Identifies the list of supported encryption types, separated by a space. The specified value is used for the default_tkt_enctypes and default_tgs_enctypes. The default encryption types, if not specified, are des-cbc-md5 and rc4-hmac. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask createKrbCo nfigFile -interactive
  ```
- Using Jython string:
  ```
  AdminTask.createKrbCon figFile ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.createKrbCon figFile ['-interactive'])
  ```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

The Kerberos configuration file

The Kerberos configuration properties, `krb5.ini` or `krb5.conf` files, must be configured on every WebSphere Application Server instance in a cell in order to use the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.

Note:

In WebSphere Application Server Version 6.1, a trust association interceptor (TAI) that uses the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) to securely negotiate and authenticate HTTP requests for secured resources was introduced. In WebSphere Application Server 7.0, this function is now deprecated. SPNEGO Web authentication has taken its place to provide dynamic reload of the SPNEGO filters and to enable fallback to the application login method.

The default Kerberos configuration file name for Windows is krb5.ini,. For other platforms is the default Kerberos configuration file name is krb5.conf,. The default location for the Kerberos configuration file is shown below:

Operating System	Default Location
Windows	c:\winnt\krb5.ini **Note:** If the **krb5.ini** file is not located in the c:\winnt directory it might be located in c:\windows directory.
Linux	/etc/krb5.conf
other UNIX-based	/etc/krb5/krb5.conf
z/OS	/etc/krb5/krb5.conf
i5/OS	/QIBM/UserData/OS400/NetworkAuthentication/krb5.conf

Note: If you don't use the default location and Kerberos configuration file name, then you have to update `*.krb5ConfigFile` properties in the soap.client.prop, ipc.client.props and sas.client.props files. Also, if the client programmatic login uses the WSKRBLogin module, you must also set the `java.security.krb5.conf` JVM property.

For SPNEGO TAI, if you don't use the default location and Kerberos configuration file name, then you must specify the `java.security.krb5.conf` JVM property.

The default Kerberos configuration file on Windows is `/winnt/krb5.ini` and on a distributed environment is `/etc/krb5`. If you specify another location path, then you must also specify the `java.security.krb5.conf` JVM property.

For example, if your `krb5.conf` file is specified at `/opt/IBM/WebSphere/profiles/AppServer/etc/krb5.conf`, then you need to specify `-Djava.security.krb5.conf=/opt/IBM/WebSphere/profiles/AppServer/etc/krb5.conf`.

The WebSphere runtime code searches for the Kerberos configuration file in the order as follows:

1. The file referenced by the Java property `java.security.krb5.conf`
2. `<java.home>/lib/security/krb5.conf`
3. `c:\winnt\krb5.ini` on Microsoft Windows platforms
4. `/etc/krb5/krb5.conf` on UNIX® platforms
5. `/etc/krb5.conf on Linux™ platforms.`

Use the wsadmin utility to configure the SPNEGO TAI for WebSphere Application Server:

1. Start WebSphere Application Server.
2. Start the command-line utility by running the **wsadmin** command from the *app_server_root*`/bin` directory from the Qshell command line.
3. At the **wsadmin** prompt, enter the following command:

   $AdminTask createKrbConfigFile

You can use the following parameters with this command:

Option	Description
<krbPath>	This parameter is required. It provides the fully qualified file system location of the Kerberos configuration (krb5.ini or krb5.conf) file.
<realm>	This parameter is required. It provides the Kerberos realm name. The value of this attribute is used by the SPNEGO TAI to form the Kerberos service principal name for each of the hosts specified with the property com.ibm.ws.security.spnego.SPN<id>.hostName.
<kdcHost>	This parameter is required. It provides the host name of the Kerberos Key Distribution Center (KDC).
<kdcPort>	This parameter is optional. It provides the port number of the KDC. The default value, if not specified, is 88.
<dns>	This parameter is required. It provides the default domain name service (DNS) that is used to produce a fully qualified host name.
<keytabPath>	This parameter is required. It provides the file system location of the Kerberos keytab file.
<encryption>	This parameter is optional. It identifies the list of supported encryption types, separated by a space. The specified value is used for the default_tkt_enctypes and default_tgs_enctypes. The default encryption types, if not specified, are des-cbc-md5 and rc4-hmac.

In the following example, the wsadmin command creates the `krb5.ini` file in the `c:\winnt` directory. The default Kerberos keytab file is also in `c:\winnt`. The actual Kerberos realm name is WSSEC.AUSTIN.IBM.COM and the KDC host name is host1.austin.ibm.com.

```
wsadmin>$AdminTask createKrbConfigFile {-krbPath
c:\winnt\krb5.ini -realm WSSEC.AUSTIN.IBM.COM -kdcHost host1.austin.ibm.com
 -dns austin.ibm.com -keytabPath c:\winnt\krb5.keytab}
```

The wsadmin command above creates a `krb5.ini` file as follows:

```
[libdefaults]
 default_realm = WSSEC.AUSTIN.IBM.COM
        default_keytab_name = FILE:c:\winnt\krb5.keytab
        default_tkt_enctypes = des-cbc-md5 rc4-hmac
        default_tgs_enctypes = des-cbc-md5 rc4-hmac
[realms]
        WSSEC.AUSTIN.IBM.COM = {
  kdc = host1.austin.ibm.com:88
              default_domain = austin.ibm.com
}
[domain_realm]
        .austin.ibm.com = WSSEC.AUSTIN.IBM.COM
```

Note:

- A Kerberos keytab file contains a list of keys that are analogous to user passwords. It is important for hosts to protect their Kerberos keytab files by storing them on the local disk. The `krb5.conf` file permission must be 644, which means that you can read and write the file; however, members of the group that the file belongs to, and all others can only read the file.
- If the run time cannot read the `default_tkt_enctypes` or `default_tgs_enctypes` entries in the `krb5.ini` file, their values are missing, or their values are not supported, the `DES-CBC-MD5` value is used by default.

The krb5.conf configuration file supports trigraphs to represent the {, }, [, and] characters. These characters depend on the language set. The natively generated keytabs cannot be read by the Kerberos client. If you have difficulty configuring SPNEGO TAI with the native krb5.conf or krb5.keytab files, complete one of the following scenarios to address the trigraphs issue:

- Replace the trigraphs in the krb5.conf file with the characters that they represent.
- Use the krb5.conf file that is generated by WebSphere Application Server.
- Use a Microsoft Windows or a key distribution center (KDC) generated keytab file.

Kerberos configuration settings, the Kerberos key distribution center (KDC) name, and realm settings for the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) are provided in the Kerberos configuration file or through `java.security.krb5.kdc` and `java.security.krb5.realm` system property files.

SPNEGO Web authentication configuration commands

Use wsadmin commands to configure, unconfigure, validate, or display Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) in the security configuration.

Configure SPNEGO Web authentication

Note: You must first have a workable Kerberos configuration file and a Kerberos keytab file. Read about Creating a Kerberos configuration file and for more information.

Use the **configureSpnego** command to configure SPNEGO as a Web authenticator in the security configuration.

At the **wsadmin** prompt, enter the following command for help:

Wsadmin>$AdminTask help configureSpnego

You can use the following parameters with the **configureSpnego** command:

Option	Description
<enabled>	This parameter is optional. It enables SPNEGO Web authentication.
<dynamicReload>	This parameter is optional. It enables dynamic reload of SPNEGO Web authentication filters.
<allowAppAuthMethodFallback>	This parameter is optional. It allows fall back to the application authentication mechanism.
<krb5Config>	This parameter is required. It supplies the directory location and file name of the configuration (krb5.ini or krb5.conf) file.
<krb5Keytab>	This parameter is optional. It supplies the directory location and file name of the Kerberos keytab file. If you do not specify this parameter, the default keytab in the Kerberos configuration file is used.

Unconfigure SPNEGO Web authentication

Use the unconfigureSpnego command to unconfigure SPNEGO Web authentication in the security configuration.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help unconfigureSpnego

Show SPNEGO Web authentication

Use the showSPNEGO command to display the SPNEGO Web authentication in the security configuration.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help showSpnego

Validate Kerberos configuration

Use the validateKrbConfig command to validate the Kerberos configuration data either in the global security file `security.xml` or specified as an input parameter.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help validateKrbConfig

You can use the following parameters with the validateKrbConfig command:

Option	Description
<checkConfigOnly>	Checks the Kerberos configuration without validating, You must use global security for this check.
<useGlobalSecurityConfig>	Uses the Global Security configuration data, security.xml, instead of input parameters.
<validateKrbRealm>	Validates the Kerberos realm against the default Kerberos realm in the Kerberos configuration file (krb5.ini or krb5.conf).
<serverId>	Specifies the server identity that is used for internal process communications.
<serverIdPassword>	Specifies the password that is used for the server identity.
<krb5Spn>	Specifies the Kerberos service principal name in the Kerberos keytab file.
<krb5Config >	This parameter is required. It supplies the directory location and file name of the configuration (krb5.ini or krb5.conf) file.
<krb5Keytab>	This parameter is optional. It supplies the directory location and file name of the Kerberos keytab file. If you do not specify this parameter, the default keytab in the Kerberos configuration file is used.
<krb5Realm >	This parameter is required. It specifies the value for the Kerberos realm name.

SPNEGO Web authentication filter commands

Use **wsadmin** commands to add, modify, delete, or show Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) Web authentication filters in the security configuration.

Add SPNEGO web authentication filter

Use the **addSpnegoFilter** command to add a new SPNEGO Web authentication filter in the security configuration.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help addSpnegoFilter

You can use the following parameters with the **addSpnegoFilter** command:

Option	Description
<hostName>	This parameter is required. Use to supply a fully-qualified host name.
<krb5Realm>	This parameter is not required. Use to supply a Kerberos realm name. If the krb5Realm parameter is not specified, the default Kerberos realm name in the Kerberos configuration file is used.
<filterCriteria>	This parameter is not required. Use to supply the HTTP request filter rules. If the filterCriteria parameter is not specified, all of the HTTP requests are authenticated by SPNEGO.
<filterClass>	This parameter is not required. Use to supply the HTTP request filter rules. If the filterClass parameter is not specified, the default filter class, com.ibm.ws.security.spnego.HTTPHeaderFilter, is used.
<trimUserName>	This parameter is not required. Use to indicate whether the Kerberos realm name is to be removed from the Kerberos principal name.
<enabledGssCredDelegate>	This parameter is not required. Use to indicate whether to extract and place the client GSS delegation credential in the subject. The default value is `true`.
<spnegoNotSupportedPage>	This parameter is not required. Use to supply the uniform resource identifier (URI) of the resource with a response to be used when SPNEGO is not supported. If this parameter is not specified, the default SPNEGO not supported error page is used.
<ntlmTokenReceivedPage>	This parameter is not required. Use to supply the URI of the resource with a response to be used when an NT LAN manager (NTLM) token is received. If this parameter is not specified, the default NTLM token received error page is used.

The following is an example of the **addSpnegoFilte**r command:

```
wsadmin>$AdminTask addSpnegoFilter {
  -hostName ks.austin.ibm.com
  -krb5Realm WSSEC.AUSTIN.IBM.COM}
```

Modify SPNEGO web authentication filter

Use the **modifySpnegoFilter** command to modify SPNEGO filter attributes in the security configuration.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help modifySpnegoFilter

You can use the following parameters with the **modifySpnegoFilter** command:

Option	Description
<hostName>	This parameter is required. Use to supply a long host name. The hostname is an identifier, so you can not modify the hostname.
<krb5Realm>	This parameter is not required. Use to supply a Kerberos realm name. If the krb5Realm parameter is not specified, the default Kerberos realm name in the Kerberos configuration file is used.
<filterCriteria>	This parameter is not required. Use to supply the HTTP request filter rules. If the filterCriteria parameter is not specified, all of the HTTP requests are authenticated by SPNEGO. **Note:** Read about Enabling and configuring SPNEGO Web authentication using the administrative console for more information about filter criteria.
<filterClass>	This parameter is not required. Use to supply the HTTP request filter rules. If the filterClass is not specified, the default filter class, com.ibm.ws.security.spnego.HTTPHeaderFilter, is used.
<trimUserName>	This parameter is not required. Use to indicate whether the Kerberos realm name is to be removed from the Kerberos principal name.
<enabledGssCredDelegate>	This parameter is not required. Use to indicate whether to extract and place the client GSS delegation credential in the subject. The default value is `true`.
<spnegoNotSupportedPage>	This parameter is not required. Use to supply the URI of the resource with a response to be used when SPNEGO is not supported. If this parameter is not specified, the default SPNEGO not supported error page is used.
<ntlmTokenReceivedPage>	This parameter is not required. Use to supply the URI of the resource with a response to be used when an NTLM token is received. If this parameter is not specified, the default NTLM token received error page is used.

The following is an example of the **modifySpnegoFilter** command:

```
wsadmin>$AdminTask modifySpnegoFilter {
  -hostName ks.austin.ibm.com
  -krb5Realm WSSEC.AUSTIN.IBM.COM}
```

Delete SPNEGO web authentication filter

Use the **deleteSpnegoFilter** command to remove SPNEGO a Web authentication filter from the security configuration. If a host name is not specified, all of the SPNEGO Web authentication filters are removed.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help deleteSpnegoFilter

You can use the following parameter with the **deleteSpnegoFilter** command:

Option	Description
<hostname>	This parameter is required. If the hostname is not specified, all of the SPNEGO Web authentication filters are deleted.

The following is an example of the **deleteSpnegoFilter** command:

```
wsadmin> $AdminTask deleteSpnegoFilter {-hostName ks.austin.ibm.com}
```

Show SPNEGO web authentication filter

Use the **showSpnegoFilter** command to display a SPNEGO Web authentication filter in the security configuration. If a host name is not specified, all of the SPNEGO filters are displayed.

At the **wsadmin** prompt, enter the following command for help:

wsadmin>$AdminTask help showSpnegoFilter

You can use the following parameter with the **showSpnegoFilter** command:

Option	Description
<hostname>	This parameter is optional. If a long host name is not specified, all of the SPNEGO Web authentication filters are displayed.

The following is an example of the **showSpnegoFilter** command:

```
wsadmin> $AdminTask showSpnegoFilter {-hostName ks.austin.ibm.com}
```

Kerberos authentication commands

Use **wsadmin** commands to create, modify or delete Kerberos as the authentication mechanism for WebSphere Application Server.

Create Kerberos authentication mechanism

Note:

The following items are required before you attempt to use the **createKrbAuthMechanism** command to create the KRB5 authentication mechanism security object field in the security configuration file:

- If you do not already have a Kerberos configuration file (**krb5.ini** or **krb5.conf**), use the **createkrbConfigFile** command task to create the Kerberos configuration file. Read about Creating a Kerberos configuration file for more information.
- You must have a Kerberos keytab file **(krb5.keytab**) that contains a Kerberos service principal name (SPN), <service name>/<fully qualified hostname>@KerberosRealm, for each machine that run WebSphere application servers. The service name can be anything; the default value is `WAS`.

  For example, if you have two application server machines, `host1.austin.ibm.com` and `host2.austin.ibm.com`, the Kerberos keytab file must contain the <service name>/ host1.austin.ibm.com and service name>/host2.austin.ibm.com SPNs and their Kerberos keys.

Use the **createKrbAuthMechanism** command to create the KRB5 authentication mechanism security object field in the security configuration file.

At the **wsadmin** prompt, enter the following command:

$AdminTask help createKrbAuthMechanism

You can use the following parameters with the **createKrbAuthMechanism** command:

Option	Description
<krb5Realm>	This parameter is optional. It indicates the Kerberos realm name. If you do not specify this parameter, the default Kerberos realm in the Kerberos configuration file is used.
<krb5Config>	This parameter is required. It indicates the directory location and file name of the configuration (krb5.ini or krb5.conf) file.
<krb5Keytab>	This parameter is optional. It indicates the directory location and file name of the Kerberos keytab file. If you do not specify this parameter, the default keytab in the Kerberos configuration file is used.
<serviceName>	This parameter is required. It indicates the Kerberos service name. The default Kerberos service name is `WAS`.
<trimUserName>	This parameter is optional. It removes the suffix of the principal user name, starting from the "@" that precedes the Kerberos realm name. This parameter is optional. The default value is `true`.
<enabledGssCredDelegate>	This parameter is not required. Use to indicate whether to extract and place the client GSS delegation credential in the subject. The default value is `true`. **Note:** If this parameter is `true`, and the runtime cannot extract the GSS delegation credential, the runtime logs a warning message.
<allowKrbAuthForCsiInbound>	This parameter is optional. It enables Kerberos authentication mechanism for Common Secure Interoperability (CSI) inbound. The default value is `true`.
<allowKrbAuthForCsiOutbound>	This parameter is required. It enables Kerberos authentication mechanism for CSI outbound. The default value is `true`.

The following is an example of the createKrbAuthMechanism command:

```
wsadmin>$AdminTask createKrbAuthMechanism {
  -krb5Realm  WSSEC.AUSTIN.IBM.COM
  -krb5Config C:\\WINNT\\krb5.ini
  -krb5Keytab C:\\WINNT\\krb5.keytab
  -serviceName WAS }
```

Modify Kerberos authentication mechanism

Use the **modifyKrbAuthMechanism** command to make changes to the KRB5 authentication mechanism security object field in the security configuration file.

At the **wsadmin** prompt, enter the following command:

wsadmin>$AdminTask help modifyKrbAuthMechanism

You can use the following parameters with the **modifyKrbAuthMechanism** command:

Option	Description
<krb5Realm>	This parameter is optional. It indicates the Kerberos realm name. If you do not specify this parameter, the default Kerberos realm in the Kerberos configuration file is used.
<krb5Config>	This parameter is required. It indicates the directory location and file name of the configuration (krb5.ini or krb5.conf) file.
<krb5Keytab>	This parameter is optional. It indicates the directory location and file name of the Kerberos keytab file. If you do not specify this parameter, the default keytab in the Kerberos configuration file is used.
<serviceName>	This parameter is required. It indicates the Kerberos service name. The default Kerberos service name is `WAS`.
<trimUserName>	This parameter is optional. It removes the suffix of the principal user name, starting from the "@" that precedes the Kerberos realm name. This parameter is optional. The default value is `true`.
<enabledGssCredDelegate>	This parameter is not required. Use to indicate whether to extract and place the client GSS delegation credential in the subject. The default value is `true`. **Note:** If this parameter is `true`, and the runtime cannot extract the GSS delegation credential, the runtime logs a warning message.
<allowKrbAuthForCsiInbound>	This parameter is optional. It enables Kerberos authentication mechanism for Common Secure Interoperability (CSI) inbound. The default value is `true`.
<allowKrbAuthForCsiOutbound>	This parameter is optional. It enables Kerberos authentication mechanism for CSI outbound. The default value is `true`.

The following is an example of the **modifyKrbAuthMechanism** command:

```
wsadmin>$AdminTask modifyKrbAuthMechanism {
   -krb5Realm  WSSEC.AUSTIN.IBM.COM
   -krb5Config C:\\WINNT\\krb5.ini
   -krb5Keytab C:\\WINNT\\krb5.keytab
   -serviceName WAS }
```

Delete Kerberos authentication mechanism

Use the **deleteKrbAuthMechanism** command to remove the KRB5 authentication mechanism security object field in the security configuration file.

At the **wsadmin** prompt, enter the following command:

wsadmin>$AdminTask help deleteKrbAuthMechanism

The following is an example of the **deleteKrbAuthMechanism** command:

```
wsadmin>$AdminTask deleteKrbAuthMechanism
```

Set active authentication mechanism

Use the **setActiveAuthMechanism** command to set the active authentication mechanism attribute in the security configuration.

At the **wsadmin** prompt, enter the following command:

wsadmin>$AdminTask help setActiveAuthMechanism

You can use the following parameter with the **setActiveAuthMechanism** command:

Option	Description
<authMechanismType>	This parameter is not required. It indicates the authentication mechanism type. The default is `KRB5`.

The following is an example of the **setActiveAuthMechanism** command:

```
wsadmin> $AdminTask setActiveAuthMechanism {-authMechanismType KRB5 }
```

Chapter 11. Configuring data access with scripting

Use these topics to learn about using scripting to configure data access.

About this task

This topic contains the following tasks:

Configuring a JDBC provider using scripting

You can configure a JDBC provider using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 77 article for more information.

1. There are two ways to perform this task. Perform one of the following:
   - Using the AdminTask object:
     - Using Jacl:
       ```
       $AdminTask createJDBCProvider {-interactive}
       ```
     - Using Jython:
       ```
       AdminTask.createJDBCProvider (['-interactive'])
       ```
   - Using the AdminConfig object:
     a. Identify the parent ID and assign it to the node variable. The following example uses the node configuration object as the parent. You can modify this example to use the cell, cluster, server, or application configuration object as the parent.
        - Using Jacl:
          ```
          set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
          ```
        - Using Jython:
          ```
          node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
          print node
          ```

        Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

b. Identify the required attributes:

Note: For supported JDBC drivers, you can also script JDBC providers according to the same pre-configured templates that are used by the administrative console logic. Consult the article “Creating configuration objects using the wsadmin tool” on page 42 for details.

– Using Jacl:

```
$AdminConfig required JDBCProvider
```

– Using Jython:

```
print AdminConfig.required('JDBCProvider')
```

Example output:

```
Attribute     Type
name     String
implementationClassName   String
```

c. Set up the required attributes and assign it to the jdbcAttrs variable. You can modify the following example to setup non-required attributes for JDBC provider.

– Using Jacl:

```
set n1 [list name JDBC1]
set implCN [list implementationClassName myclass]
set  jdbcAttrs [list  $n1  $implCN]
```

Example output:

```
{name {JDBC1}} {implementationClassName {myclass}}
```

– Using Jython:

```
n1 = ['name', 'JDBC1']
implCN = ['implementationClassName', 'myclass']
jdbcAttrs = [n1,  implCN]
print jdbcAttrs
```

Example output:

```
[['name', 'JDBC1'], ['implementationClassName', 'myclass']]
```

d. Create a new JDBC provider using node as the parent:

– Using Jacl:

```
$AdminConfig create JDBCProvider $node $jdbcAttrs
```

– Using Jython:

```
AdminConfig.create('JDBCProvider', node, jdbcAttrs)
```

Example output:

```
JDBC1(cells/mycell/nodes/mynode|resources.xml#JDBCProvider_1)
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

What to do next

If you modify the class path or native library path of a JDBC provider: After saving your changes (and synchronizing the node in a network deployment environment), you must restart every application server within the scope of that JDBC provider for the new configuration to work. Otherwise, you receive a data source failure message.

Configuring new data sources using scripting

You can configure new data sources using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

In WebSphere Application Server, any JDBC driver properties that are required by your database vendor must be set as data source properties. Consult the article Vendor-specific data sources minimum required settings in the *Troubleshooting and support* PDF to see a list of these properties and setting options, ordered by JDBC provider type. Consult your database vendor documentation to learn about available optional data source properties. Script them as *custom properties* after you create the data source. In the Related links section of this article, click the ″Configuring new data source custom properties using scripting″ link for more information.

About this task

There are two ways to perform this task; use either of the following wsadmin scripting objects:

- AdminTask object
- AdminConfig object

AdminConfig gives you more configuration control than the AdminTask object. When you create a data source using AdminTask, you supply universally required properties only, such as a JNDI name for the data source. (Consult the article “JDBCProviderManagement command group for the AdminTask object” on page 965 for more information.) Other properties that are required by your JDBC driver are assigned default values by Application Server. You cannot use AdminTask commands to set or edit these properties; you must use AdminConfig commands.

- Using the AdminConfig object to configure a new data source:
  1. Identify the parent ID, which is the name and location of the JDBC provider that supports your data source.
     - Using Jacl:

       ```
       set newjdbc [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/]
       ```

     - Using Jython:

       ```
       newjdbc = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/')
       print newjdbc
       ```

     Example output:

     ```
     JDBC1(cells/mycell/nodes/mynode|resources.xml#JDBCProvider_1)
     ```

  2. Obtain the required attributes.

     Note: For supported JDBC drivers, you can also script data sources according to the same pre-configured templates that are used by the administrative console logic. Consult the article “Creating configuration objects using the wsadmin tool” on page 42 for details.

     - Using Jacl:

       ```
       $AdminConfig required DataSource
       ```

     - Using Jython:

       ```
       print AdminConfig.required('DataSource')
       ```

     Example output:

     ```
     Attribute   Type
     name    String
     ```

     Note: If the database vendor-required properties (which are referenced in the article Data source minimum required settings, by vendor) are not displayed in the resulting list of required attributes, script these properties as data source custom properties after you create the data source.

3. Set up the required attributes.
   - Using Jacl:
     ```
     set name [list name DS1]
     set dsAttrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'DS1']
     dsAttrs = [name]
     ```
4. Create the data source.
   - Using Jacl:
     ```
     set newds [$AdminConfig create DataSource $newjdbc $dsAttrs]
     ```
   - Using Jython:
     ```
     newds = AdminConfig.create('DataSource', newjdbc, dsAttrs)
     print newds
     ```

   Example output:

   ```
   DS1(cells/mycell/nodes/mynode|resources.xml#DataSource_1)
   ```

- Using the AdminTask object to configure a new data source:
  - Using Jacl:
    ```
    $AdminTask createDatasource {-interactive}
    ```
  - Using Jython:
    ```
    AdminTask.createDatasource (['-interactive'])
    ```
- Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
- Synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

What to do next

To set additional properties that are supported by your JDBC driver, script them as data source custom properties.

Configuring new connection pools using scripting

You can use scripting and the wsadmin tool to configure new connection pools.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
     ```
   - Using Jython:
     ```
     newds = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/')
     ```

   Example output:

   ```
   DS1(cells/mycell/nodes/mynode|resources.xml$DataSource_1)
   ```
2. Creating connection pool:

- Using Jacl:

  ```
  $AdminConfig create ConnectionPool $newds {}
  ```

- Using Jython:

  ```
  print AdminConfig.create('ConnectionPool', newds, [])
  ```

  Example output:

  ```
  (cells/mycell/nodes/mynode|resources.xml#ConnectionPool_1)
  ```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Changing connection pool settings with the wsadmin tool

You can use the wsadmin scripting tool to change connection pool settings.

About this task

The WebSphere Application Server wsadmin tool provides the ability to run scripts. You can use the wsadmin tool to manage a WebSphere Application Server installation, as well as configuration, application deployment, and server run-time operations. The WebSphere Application Server only supports the Jacl and Jython scripting languages. To learn more about the wsadmin tool see Starting the wsadmin scripting client.

To use the wsadmin tool to change connection pool settings:

1. Launch a scripting command. There are several options for you to run scripting commands, ranging from running them interactively to running them in a profile.

   To change connection pool settings, you use the *getAttribute* and *setAttribute* commands, run against the various settings objects.

   For example, to change the connection timeout setting, the commands are:

   ```
   $AdminControl getAttribute $objectname connectionTimeout
   $AdminControl setAttribute $objectname connectionTimeout 200
   ```

   where:
   - $ is a Jacl operator for substituting a variable name with its value
   - getAttribute and setAttribute are the commands
   - connectionTimeout is the object whose value you are resetting
2. For Jacl code examples of each of the connection pool settings, see “Example: Changing connection pool settings with the wsadmin tool”

Example: Changing connection pool settings with the wsadmin tool

Using the wsadmin AdminControl object, you can script changes to connection pool settings.

The WebSphere Application Server wsadmin tool provides the ability to run scripts in the Jacl and Jython languages only. You must start the wsadmin scripting client to perform any scripting task. (See the article ″Starting the wsadmin scripting client.″)

For more information about the AdminControl scripting object, refer to the article ″Using the AdminControl object for scripted administration.″ See the links section at the end of this article.

Connection Timeout

The Connection timeout can be changed at any time while the pool is active. All connection requests that are waiting for a connection are changed to the new value minus the time already waited, and are returned to the wait state if there are no available connections.

For example, if the connection timeout is changed to 300 seconds and a connection request has waited 100 seconds, the connection request waits for 200 more seconds if no connection becomes available.

```
$AdminControl getAttribute $objectname connectionTimeout
$AdminControl setAttribute $objectname connectionTimeout 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Maximum Connections

The maximum connections can be changed at any time except in the case where stuck connection support is active.

If stuck connection support is active, an attempt to change the maximum connections is made. If the attempt fails, an `IllegalState` exception occurs. See the Connection pool advanced settings topic in the *Administering applications and their environment* PDF for more information.

If stuck connection support is not active, the maximum connections are changed to the new value. If the new value is greater than the current value, the number of connections increases to the new value and any requests waiting are notified. If the new value is less than the current value and Aged Timeout or Reap Time is used, the number of connections decreases to the new value depending on pool activity. If agedTimeout or Reap Time are not used, no automatic attempt will be made to reduce the total number of connections. To manually reduce the number of connections to the new maximum connections, use the Mbean function purgePoolContents.

```
$AdminControl getAttribute $objectname maxConnections
$AdminControl setAttribute $objectname maxConnections 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Minimum Connections

The minimum connections can be changed at any time

```
$AdminControl getAttribute $objectname minConnections
$AdminControl setAttribute $objectname minConnections 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Reap Time

The reap time can be changed at any time. The reap time interval is changed to the new value at the next interval.

```
$AdminControl getAttribute $objectname reapTime
$AdminControl setAttribute $objectname reapTime 30
```

Unused Timeout

The unused timeout can be changed at any time.

```
$AdminControl getAttribute $objectname unusedTimeout
$AdminControl setAttribute $objectname unusedTimeout 900
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Aged Timeout

The Aged Timeout can be changed at any time.

```
$AdminControl getAttribute $objectname agedTimeout
$AdminControl setAttribute $objectname agedTimeout 900
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Purge Policy

The purge policy can be changed at any time.

```
$AdminControl getAttribute $objectname purgePolicy
$AdminControl setAttribute $objectname purgePolicy "Failing Connection Only"
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Surge Protection Support

Surge connection support starts if surgeThreshold is > -1 and surgeCreationInterval is > 0. The surge protection properties can be changed at any time.

```
$AdminControl getAttribute $objectname surgeCreationInterval
$AdminControl setAttribute $objectname surgeCreationInterval 30
$AdminControl getAttribute $objectname surgeThreshold
$AdminControl setAttribute $objectname surgeThreshold 15
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Stuck connection Support

An attempt is made to change the stuckTime, stuckTimerTime or stuckThreshold properties. If the attempt fails, an `IllegalState` exception occurs. The pool cannot have any active requests or active connections during this request. For the stuck connection support to start, all three stuck property values must be greater than 0, and the maximum connections value must be > 0.

If the connection pool is stuck, you cannot change the stuck or the maximum connection properties. If you are stuck, there are active connections.

```
$AdminControl getAttribute $objectname stuckTime
$AdminControl setAttribute $objectname stuckTime 30
$AdminControl getAttribute $objectname stuckTimerTime
$AdminControl setAttribute $objectname stuckTimerTime 15
$AdminControl getAttribute $objectname stuckThreshold
$AdminControl setAttribute $objectname stuckThreshold 10
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Test Connection Support (This is only supported for a DataSource)

The test connection support starts when the testConnection property is set to true, and the interval is > 0. The test connection properties can be changed at any time.

```
$AdminControl getAttribute $objectname testConnection
$AdminControl setAttribute $objectname testConnection 30
$AdminControl getAttribute $objectname testConnectionInterval
$AdminControl setAttribute $objectname testConnectionInterval 15
```

For more information about this setting, see the Test connection service topic in the *Administering applications and their environment* PDF.

Connection Pool Partition Support

```
$AdminControl invoke $objectname freePoolDistributionTableSize
$AdminControl invoke $objectname numberOfFreePoolPartitions
$AdminControl invoke $objectname numberOfSharedPoolPartitions
$AdminControl invoke $objectname gatherPoolStatisticalData
$AdminControl invoke $objectname enablePoolStatisticalData
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Show Pool Information Support

The show pool operations can be changed at any time.

```
$AdminControl invoke $objectname showAllPoolContents
$AdminControl invoke $objectname showPoolContents
$AdminControl invoke $objectname showAllocationHandleList
```

Purge Connection Support

PurgePool can be changed at any time.

```
$AdminControl invoke $objectname purgePoolContents normal
$AdminControl invoke $objectname purgePoolContents immediate
```

Pool Control Support

Pause and resume can be changed at any time.

```
$AdminControl invoke $objectname pause
$AdminControl invoke $objectname resume
```

Example: Accessing MBean connection factory and data sources using wsadmin

Wsadmin commands are used to access connection factory and data source MBeans. MBeans can retrieve or update properties for a connection factory or data source.

To get a list of J2C connection factory or data source Mbean object names, from the wsadmin command line use one of the following administrative control commands:

```
$AdminControl queryNames *:type=J2CConnectionFactory,*
$AdminControl queryNames *:type=DataSource,*
```

Example output from DataSource query:

```
wsadmin>$AdminControl queryNames *:type=DataSource,*
"WebSphere:name=Default Datasource,process=server1,platform=dynamicproxy,node=
system1Node01,JDBCProvider=Derby JDBC Provider,j2eeType=JDBCDataSource,J2EESe
rver=server1,Server=server1,version=6.0.0.0,type=DataSource,mbeanIdentifier=cell
s/system1Node01Cell/nodes/system1Node01/servers/server1/resources.xml#DataSource
_1094760149902,JDBCResource=Derby JDBC Provider,cell=system1Node01Cell"
```

By using the J2C connection factory or data source MBean object name, you can access the MBean. The name follows the `webSphere:name=` expression. In the following example, for the first set, the default data source name is set on variable name. For the second set, the object name is set on variable *objectName*.

Example using the Default Datasource

```
- wsadmin>set name [list Default Datasource]

Default Datasource


- wsadmin>set objectName [$AdminControl queryNames *:name=$name,*]


"WebSphere:name=Default Datasource,process=server1,platform=dynamicproxy,node=
system1Node01,JDBCProvider=Derby JDBC Provider,j2eeType=JDBCDataSource,J2EESe
rver=server1,Server=server1,version=6.0.0.0,type=DataSource,mbeanIdentifier=cell
s/system1Node01Cell/nodes/system1Node01/servers/server1/resources.xml#DataSource
_1094760149902,JDBCResource=Derby JDBC Provider,cell=system1Node01Cell"
```

Now you can use the objectName for getting help on attributes and operations available for this mbean.

```
$Help attributes $objectName
$Help operations $objectName
```

To get or set an attribute or to use an operation, use one of the following commands:

```
$AdminControl getAttribute $objectName "attribute name"
$AdminControl setAttribute $objectName "attribute name" value
$AdminControl invoke $objectName "operation"
```

Attribute, operation examples:

```
Get and set a attribute maxConnections (Note, if you get no value, a null value, or
a java.lang.IllegalStateException, the connection factory or data source for this
MBean has not been created. The connection factory or data source is created at first
JNDI lookup. For this example, the Default Datasource needs to be used before get,
set and invoke will work.)

wsadmin>$AdminControl getAttribute $objectName maxConnections
10

wsadmin>$AdminControl setAttribute $objectName maxConnections 20

wsadmin>$AdminControl getAttribute $objectName maxConnections
20

Using invoke

wsadmin>$AdminControl invoke $objectName showPoolContents

PoolManager name:DefaultDatasource
PoolManager object:746354514
Total number of connections: 3 (max/min 20/1, reap/unused/aged 180/1800/0,
connectiontimeout/purge 1800/EntirePool)
                            (testConnection/inteval false/0, stuck timer/time
/threshold 0/0/0, surge time/connections 0/-1)
Shared Connection information (shared partitions 200)
  No shared connections

Free Connection information (free distribution table/partitions 5/1)
  (2)(0)MCWrapper id 5c6af75b  Managed connection WSRdbManagedConnectionImpl@4b5
a775b  State:STATE_ACTIVE_FREE
  (2)(0)MCWrapper id 3394375a  Managed connection WSRdbManagedConnectionImpl@328
5f75a  State:STATE_ACTIVE_FREE
  (2)(0)MCWrapper id 4795b75a  Managed connection WSRdbManagedConnectionImpl@46a
4b75a  State:STATE_ACTIVE_FREE

  Total number of connection in free pool: 3
UnShared Connection information
  No unshared connections
```

Here is an example Java program using the wsadmin tool to invoke showPoolContents: the Example: using wsadmin to invoke showPoolContents topic in the *Administering applications and their environment* PDF.

Example: Invoking showPoolContents using the wsadmin tool

This example Java program uses the wsadmin tool to invoke showPoolContents.

```
/**
 * "This sample program is provided AS IS and may be used, executed, copied and modified without royalty payment
 *  by customer (a) for its own instruction and study, (b) in order to develop applications designed to run with
 *  an IBM WebSphere product, either for customer's own internal use or for redistribution by customer, as part
 *  of such an application, in customer's own products. "

 * Product 5724i63, (C) COPYRIGHT International Business Machines Corp., 2004


 *
 * All Rights Reserved * Licensed Materials - Property of IBM
 *
 * This program will display an active mbean object name and the connection pool.
 *
 * To run this program
 *
 *  1. call "%WAS_HOME%/bin/setupCmdLine.bat"
 *
 *  2. "%JAVA_HOME%\bin\java" "%CLIENTSAS%" "-Dwas.install.root=%WAS_HOME%" "-Dwas.repository.root=%CONFIG_ROOT%"
 *     -Dcom.ibm.CORBA.BootstrapHost=%COMPUTERNAME% -classpath "%WAS_CLASSPATH%;%WAS_HOME%\runtimes
\com.ibm.ws.admin.client_7.0.0.jar;%WAS_HOME%
 *   \lib\wasjmx.jar;." ShowPoolContents DataSource_mbean_name
 */

import java.util.Properties;
import java.util.Set;

import javax.management.*;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.exception.ConnectorException;

public class ShowPoolContents {

  private AdminClient adminClient;

  public static void main(String[] args) {
    try {
      String name2 = null;
      String port = null;
      if (args.length < 2) {
        System.out.println("Enter name for the mbean and port");
        return;
      }
      else {
        name2 = args[0];
        port = args[1];
        System.out.println("Searching for name" + name2);
      }
      ShowPoolContents ace = new ShowPoolContents();

      // Create an AdminClient
      ace.createAdminClient(port);

      ObjectName[] cfON = ace.queryMBean("DataSource", null);

      if (cfON.length == 0) {
        System.out.println(" *Error : queryMBean did not find any active mbeans of type DataSource.");
        System.out.println(" At first touch of a DataSource, an mbean will be created.
To touch a DataSource, use getConnection");
        return;
      }
      int selectedObjectName = -1;
      String findName = name2;

      for (int i = 0; i < cfON.length; i++) {
        System.out.println("\ncfON[i] = " + cfON[i]);
      }

    }
    catch (Exception e) {
      System.out.println(" *Exception : " + e.toString());
      e.printStackTrace(System.out);
    }
  }

  private void createAdminClient(String port) {
    // Set up a Properties object for the JMX connector attributes
    Properties connectProps = new Properties();
```

```
    connectProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
    connectProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
    connectProps.setProperty(AdminClient.CONNECTOR_PORT, port);

    // Get an AdminClient based on the connector properties
    try {
      adminClient = AdminClientFactory.createAdminClient(connectProps);
    }
    catch (ConnectorException e) {
      System.out.println("Exception creating admin client: " + e);
      System.exit(-1);
    }

    System.out.println("Connected to DeploymentManager");
  }

  // helper method to query mbean by type / name
  public ObjectName[] queryMBean(String type, String name) throws Exception {
    String s = "*:";
    if (type != null)
      s += "type=" + type;

    if (name != null)
      s += ",name=" + name;

    s += ",*";

    System.out.println("queryMBean: " + s);
    ObjectName ion = new ObjectName(s);

    Set set = adminClient.queryNames(ion, null);
    Object[] o = set.toArray();
    ObjectName[] on = new ObjectName[o.length];

    for (int i = 0; i < o.length; i++) {
      on[i] = (ObjectName)o[i];
    }

    return on;
  }
}
```

Related tasks

"Changing connection pool settings with the wsadmin tool" on page 943
You can use the wsadmin scripting tool to change connection pool settings.

Configuring new data source custom properties using scripting

You can configure new data source custom properties using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 77 article for more information.

About this task

Perform the following steps to configure a new data source custom property:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
     ```
   - Using Jython:

     ```
     newds = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/')
     print newds
     ```

     Example output:

     ```
     DS1(cells/mycell/nodes/mynode|resources.xml$DataSource_1)
     ```
2. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newds propertySet]
     ```
   - Using Jython:

```
propSet = AdminConfig.showAttribute(newds, 'propertySet')
print propSet
```

Example output:

```
(cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_8)
```

3. Get required attribute:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute     Type
   name         String
   ```

4. Set up attributes:
   - Using Jacl:

     ```
     set name [list name RP4]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP4']
     rpAttrs = [name]
     ```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP4(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_8)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new Java 2 Connector authentication data entries using scripting

You can configure new Java 2 Connector (J2C) authentication data entries with the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new J2C authentication data entry:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set security [$AdminConfig getid /Cell:mycell/Security:/]
     ```

   - Using Jython:

```
security = AdminConfig.getid('/Cell:mycell/Security:/')
print security
```

Example output:

```
(cells/mycell|security.xml#Security_1)
```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required JAASAuthData
     ```

   - Using Jython:

     ```
     print AdminConfig.required('JAASAuthData')
     ```

   Example output:

   ```
   Attribute       Type
   alias           String
   userId          String
   password        String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set alias [list alias myAlias]
     set userid [list userId myid]
     set password [list password secret]
     set jaasAttrs [list $alias $userid $password]
     ```

     Example output:

     ```
     {alias myAlias} {userId myid} {password secret}
     ```

   - Using Jython:

     ```
     alias = ['alias', 'myAlias']
     userid = ['userId', 'myid']
     password = ['password', 'secret']
     jaasAttrs = [alias, userid, password]
     print jaasAttrs
     ```

     Example output:

     ```
     [['alias', 'myAlias'], ['userId', 'myid'], ['password', 'secret']]
     ```

4. Create JAAS auth data:
   - Using Jacl:

     ```
     $AdminConfig create JAASAuthData $security $jaasAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('JAASAuthData', security, jaasAttrs)
     ```

   Example output:

   ```
   (cells/mycell|security.xml#JAASAuthData_2)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WAS40 data sources using scripting

Use scripting to configure a new WAS40 data source.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newjdbc [$AdminConfig getid "/JDBCProvider:Apache Derby JDBC Provider/"]
     ```
   - Using Jython:
     ```
     newjdbc = AdminConfig.getid('/JDBCProvider:Apache Derby JDBC Provider/')
     print newjdbc
     ```

   Example output:

   ```
   JDBC1(cells/mycell/nodes/mynode|resources.xml$JDBCProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required WAS40DataSource
     ```
   - Using Jython:
     ```
     print AdminConfig.required('WAS40DataSource')
     ```

   Example output:

   ```
   Attribute    Type
   name    String
   ```
3. Set up required attributes:
   - Using Jacl:
     ```
     set name [list name was4DS1]
     set ds4Attrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'was4DS1']
     ds4Attrs = [name]
     ```
4. Create WAS40DataSource:
   - Using Jacl:
     ```
     set new40ds [$AdminConfig create WAS40DataSource $newjdbc $ds4Attrs]
     ```
   - Using Jython:
     ```
     new40ds = AdminConfig.create('WAS40DataSource', newjdbc, ds4Attrs)
     print new40ds
     ```

   Example output:

   ```
   was4DS1(cells/mycell/nodes/mynode|resources.xml#WAS40DataSource_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WAS40 connection pools using scripting

You can use scripting to configure a new WAS40 connection pool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WAS40 connection pool:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/
     Server:server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
     ```

   - Using Jython:

     ```
     new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/
     Server:server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')
     print new40ds
     ```

   Example output:

   ```
   was4DS1(cells/mycell/nodes/mynodes:resources.xml$WAS40DataSource_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WAS40ConnectionPool
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WAS40ConnectionPool')
     ```

   Example output:

   ```
   Attribute       Type
   minimumPoolSize     Integer
   maximumPoolSize     Integer
   connectionTimeout   Integer
   idleTimeout        Integer
   orphanTimeout       Integer
   statementCacheSize  Integer
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set mps [list minimumPoolSize 5]
     set minps [list minimumPoolSize 5]
     set maxps [list maximumPoolSize 30]
     set conn [list connectionTimeout 10]
     set idle [list idleTimeout 5]
     set orphan [list orphanTimeout 5]
     set scs [list statementCacheSize 5]
     set 40cpAttrs [list $minps $maxps $conn $idle $orphan $scs]
     ```

     Example output:

     ```
     {minimumPoolSize 5} {maximumPoolSize 30}
     {connectionTimeout 10} {idleTimeout 5}
     {orphanTimeout 5} {statementCacheSize 5}
     ```

   - Using Jython:

     ```
     minps = ['minimumPoolSize', 5]
     maxps = ['maximumPoolSize', 30]
     conn = ['connectionTimeout', 10]
     idle = ['idleTimeout', 5]
     orphan = ['orphanTimeout', 5]
     scs = ['statementCacheSize', 5]
     cpAttrs = [minps, maxps, conn, idle, orphan, scs]
     print cpAttrs
     ```

     Example output:

     ```
     [[minimumPoolSize, 5], [maximumPoolSize, 30],
     [connectionTimeout, 10], [idleTimeout, 5],
     [orphanTimeout, 5], [statementCacheSize, 5]]
     ```

4. Create was40 connection pool:
   - Using Jacl:

```
$AdminConfig create WAS40ConnectionPool $new40ds $40cpAttrs
```

- Using Jython:

```
print AdminConfig.create('WAS40ConnectionPool', new40ds, 40cpAttrs)
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#WAS40ConnectionPool_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WAS40 custom properties using scripting

You can use scripting and the wsadmin tool to configure a new WAS40 custom property.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WAS40 custom properties:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/
     JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
     ```

   - Using Jython:

     ```
     new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/
     JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')
     print new40ds
     ```

   Example output:

   ```
   was4DS1(cells/mycell/nodes/mynodes|resources.xml$WAS40DataSource_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newds propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newds, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_9)
   ```

3. Get required attribute:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute    Type
   name         String
   ```

4. Set up required attributes:
   - Using Jacl:

```
set name [list name RP5]
set rpAttrs [list $name]
```

- Using Jython:

```
name = ['name', 'RP5']
rpAttrs = [name]
```

5. Create J2EE Resource Property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP5(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_9)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new J2C resource adapters using scripting

Use the wsadmin tool to configure resource adapters with Resource Adapter Archive (RAR) files. A RAR file provides the classes and other code to support a resource adapter for access to a specific enterprise information system (EIS), such as the Customer Information Control System (CICS). Configure resource adapters for an EIS only after you install the appropriate RAR file.

Before you begin

A RAR file, which is often called a J2EE Connector Architecture (JCA) connector, must comply with the JCA Specification. Meet these requirements by using a supported assembly tool (as described in the Assembly tools article) to assemble a collection of Java archive (JAR) files, other runnable components, utility classes, and so on, into a deployable RAR file. Then you are ready to install your RAR file in Application Server.

There are two ways to complete this task. This topic uses the AdminConfig object to install resource adapters. Alternatively, you can use the installJ2CResourceAdapter script in the AdminJ2C script library to install a J2C resource adapter in your configuration, as the following example demonstrates:

```
AdminJ2C.installJ2CResourceAdapter("myNode", "C:\temp\jca15cmd.rar", "J2CTest")
```

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the configuration ID of the node to which the resource adapter is installed, as the following examples demonstrate:
   - Using Jacl:

     ```
     set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
     ```

   - Using Jython:

     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```

3. Identify the optional attributes.

The J2CResourceAdapter object does not require specific arguments. Use the following command to display the optional attributes for the J2CResourceAdapter object:

- Using Jacl:

```
$AdminConfig defaults J2CResourceAdapter
```

- Using Jython:

```
print AdminConfig.defaults('J2CResourceAdapter')
```

The following displays the command output that displays each optional attribute and the data type for the attribute, and denotes the default attributes:

```
Attribute                       Type
name                            String
description                     String
classpath                       String
nativepath                      String
providerType                    String
archivePath                     String
threadPoolAlias                 String
propertySet                     J2EEResourcePropertySet
jaasLoginConfiguration          JAASConfigurationEntry
deploymentDescriptor            Connector
connectionDefTemplateProps      ConnectionDefTemplateProps
activationSpecTemplateProps     ActivationSpecTemplateProps
j2cAdminObjects                 J2CAdminObject
adminObjectTemplateProps        AdminObjectTemplateProps
j2cActivationSpec               J2CActivationSpec
```

4. Set up the attributes of interest.

   Determine the attributes to configure for the J2C resource adapter. In the following examples, the commands set the RAR file path to the `rarFile` variable and the name and description configuration options to the `option` variable:

   - Using Jacl:

```
set rarFile /currentScript/cicseci.rar
set option {-rar.name  RAR1 -rar.desc "New resource adapter"}
```

   - Using Jython:

5. Create a resource adapter.

   Use the installResourceAdapter command for the AdminConfig object to install the resource adapter with the previously set configuration options, as the following examples demonstrate:

   - Using Jacl:

```
$AdminConfig installResourceAdapter $rarFile mynode $option
```

   - Using Jython:

```
AdminConfig.installResourceAdapter(rarFile, 'mynode', option)
```

   Example output:

```
RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

6. Save the configuration changes.
7. Synchronize the node.

   Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

   - Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

   - Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

```
AdminNodeManagement.syncNode("myNode")
```

Configuring custom properties for J2C resource adapters using scripting

You can configure custom properties for J2C resource adapters with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new custom property for a J2C resource adapters:

1. Identify the parent ID and assign it to the newra variable.
   - Using Jacl:
     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```
   - Using Jython:
     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

   Example output:
   ```
   RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
   ```
2. Get the J2EE resource property set:
   - Using Jacl:
     ```
     set propSet [$AdminConfig showAttribute $newra propertySet]
     ```
   - Using Jython:
     ```
     propSet = AdminConfig.showAttribute(newra, 'propertySet')
     print propSet
     ```

   Example output:
   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_8)
   ```
3. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required J2EEResourceProperty
     ```
   - Using Jython:
     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:
   ```
   Attribute     Type
   name         String
   ```
4. Set up the required attributes:
   - Using Jacl:
     ```
     set name [list name RP4]
     set rpAttrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'RP4']
     rpAttrs = [name]
     ```
5. Create a J2EE resource property:
   - Using Jacl:
     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

Example output:

```
RP4(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_8)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new J2C connection factories using scripting

Use scripting and the wsadmin tool to configure new J2C connection factories.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new J2C connection factory:

1. Identify the parent ID and assign it to the newra variable.
   - Using Jacl:

     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```
   - Using Jython:

     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

   Example output:

   ```
   RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
   ```
2. There are two ways to configure a new J2C connection factory. Perform one of the following:
   - Using the AdminTask object:
     a. List the connection factory interfaces:
        – Using Jacl:

          ```
          $AdminTask listConnectionFactoryInterfaces $newra
          ```
        – Using Jython:

          ```
          AdminTask.listConnectionFactoryInterfaces(newra)
          ```

          Example output:

          ```
          javax.sql.DataSource
          ```
     b. Create a J2CConnectionFactory:
        – Using Jacl:

          ```
          $AdminTask createJ2CConnectionFactory $newra { -name cf1
          -jndiName eis/cf1 -connectionFactoryInterface
          avax.sql.DataSource
          ```
        – Using Jython:

          ```
          AdminTask.createJ2CConnectionFactory(newra, '['-name', 'cf1',
          '-jndiName', 'eis/cf1', '-connectionFactoryInterface',
          'avax.sql.DataSource']')
          ```
   - Using the AdminConfig object:
     a. Identify the required attributes:
        – Using Jacl:

          ```
          $AdminConfig required J2CConnectionFactory
          ```
        – Using Jython:

```
print AdminConfig.required('J2CConnectionFactory')
```

Example output:

```
Attribute Type
connectionDefinition ConnectionDefinition@
```

b. If your resource adapter is JCA1.5 and you have multiple connection definitions defined, it is required that you specify the ConnectionDefinition attribute. If your resource adapter is JCA1.5 and you have only one connection definition defined, it will be picked up automatically. If your resource adapter is JCA1.0, you do not need to specify the ConnectionDefinition attribute. Perform the following command to list the connection definitions defined by the resource adapter:

– Using Jacl:

```
$AdminConfig list ConnectionDefinition $newra
```

– Using Jython:

```
print AdminConfig.list('ConnectionDefinition', $newra)
```

c. Set up the required attributes:

– Using Jacl:

```
set name [list name J2CCF1]
set jname [list jndiName eis/j2ccf1]
set j2ccfAttrs [list $name]
```

– Using Jython:

```
name = ['name', 'J2CCF1']
jname = ['jndiName', 'eis/j2ccf1']
j2ccfAttrs = [name,jname]
```

d. If you are specifying the ConnectionDefinition attribute, also set up the following:

– Using Jacl:

```
set cdattr [list connectionDefinition $cd]
```

– Using Jython:

```
cdattr = ['connectionDefinition', $cd]
```

e. Create a J2C connection factory:

– Using Jacl:

```
$AdminConfig create J2CConnectionFactory $newra $j2ccfAttrs
```

– Using Jython:

```
print AdminConfig.create('J2CConnectionFactory', newra, j2ccfAttrs)
```

Example output:

```
J2CCF1(cells/mycell/nodes/mynode|resources.xml#J2CConnectionFactory_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new J2C activation specifications using scripting

You can configure new J2C activation specifications using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a J2C activation specifications:

1. Identify the parent ID and assign it to the `newra` variable.
   - Using Jacl:

     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```
   - Using Jython:

     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

     Example output:

     ```
     RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
     ```
2. There are two ways to configure a new J2C administrative object. Perform one of the following:
   - Using the AdminTask object:
     a. List the administrative object interfaces:

        Using Jacl:

        ```
        $AdminTask listMessageListenerTypes $newra
        ```

        Using Jython:

        ```
        AdminTask.listMessageListenerTypes(newra)
        ```

        Example output:

        ```
        javax.jms.MessageListener
        ```

     b. Create a J2C administrative object:

        Using Jacl:

        ```
        $AdminTask createJ2CActivationSpec $newra { -name ac1
        -jndiName eis/ac1 -messageListenerType
        javax.jms.MessageListener}
        ```

        Using Jython:

        ```
        AdminTask.createJ2CActivationSpec(newra, ['-name', 'ao1',
        '-jndiName', 'eis/ao1', '-messageListenerType',
        'javax.jms.MessageListener'])
        ```
   - Using the AdminConfig object:
     a. Using Jacl:

        ```
        $AdminConfig required J2CActivationSpec
        ```

        Using Jython:

        ```
        print AdminConfig.required('J2CActivationSpec')
        ```

        Example output:

        ```
        Attribute Type
        activationSpec ActivationSpec@
        ```

     b. If your resource adapter is JCA V1.5 and you have multiple activation specifications defined, it is required that you specify the activation specification attribute. If your resource adapter is JCA V1.5 and you have only one activation specification defined, it will be picked up automatically. If your resource adapter is JCA V1.0, you do not need to specify the activationSpec attribute. Perform the following command to list the activation specifications defined by the resource adapter:

        Using Jacl:

        ```
        $AdminConfig list ActivationSpec $newra
        ```

        Using Jython:

        ```
        print AdminConfig.list('ActivationSpec', $newra)
        ```

     c. Set the administrative object that you need to a variable:

        Using Jacl:

```
set ac [$AdminConfig list ActivationSpec $newra]
set name [list name J2CAC1]
set jname [list jndiName eis/J2CAC1]
set j2cacAttrs [list $name $jname $cdcttr]
```

Using Jython:

```
ac = AdminConfig.list('ActivationSpec', $newra)
name = ['name', 'J2CAC1']
jname = ['jndiName', 'eis/j2cac1']
j2cacAttrs = [name, jname,cdattr]
```

d. If you are specifying the ActivationSpec attribute, also set up the following:

Using Jacl:

```
set cdcttr [list activationSpec $ac]
```

Using Jython:

```
cdattr = ['activationSpec', ac]
```

e. Create a J2C activation specification object:

Using Jacl:

```
$AdminConfig create J2CActivationSpec $newra $j2cacAttrs
```

Using Jython:

```
print AdminConfig.create('J2CActivationSpec', newra,j2cacAttrs)
```

Example output:

```
J2CAC1(cells/mycell/nodes/mynode|resources.xml#J2CActivationSpec_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new J2C administrative objects using scripting

You can use scripting and the wsadmin tool to configure new J2C administrative objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a J2C administrative object:

1. Identify the parent ID and assign it to the `newra` variable.
   - Using Jacl:

     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```

   - Using Jython:

     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

   Example output:

   ```
   RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
   ```

2. There are two ways to configure a new J2C administrative object. Perform one of the following:
   - Using the AdminTask object:
     a. List the administrative object interfaces:

        Using Jacl:

        ```
        $AdminTask listAdminObjectInterfaces $newra
        ```

Using Jython:

```
AdminTask.listAdminObjectInterfaces(newra)
```

Example output:

```
com.ibm.test.message.FVTMessageProvider
```

b. Create a J2C administrative object:

Using Jacl:

```
$AdminTask createJ2CAdminObject $newra { -name ao1 -jndiName eis/ao1
-adminObjectInterface com.ibm.test.message.FVTMessageProvider }
```

Using Jython:

```
AdminTask.createJ2CAdminObject(newra, ['-name', 'ao1', '-jndiName', 'eis/ao1',
'-adminObjectInterface', 'com.ibm.test.message.FVTMessageProvider'])
```

- Using the AdminConfig object:

a. Using Jacl:

```
$AdminConfig required J2CAdminObject
```

Using Jython:

```
print AdminConfig.required('J2CAdminObject')
```

Example output:

```
Attribute Type
adminObject AdminObject@
```

b. If your resource adapter is JCA V1.5 and you have multiple administrative objects defined, it is required that you specify the administrative object attribute. If your resource adapter is JCA V1.5 and you have only one administrative object defined, it will be picked up automatically. If your resource adapter is JCA V1.0, you do not need to specify the administrative object attribute. Perform the following command to list the administrative objects defined by the resource adapter:

Using Jacl:

```
$AdminConfig list AdminObject $newra
```

Using Jython:

```
print AdminConfig.list('AdminObject', $newra)
```

c. Set the administrative objects that you need to a variable:

Using Jacl:

```
set ao AdminObjectId
set name [list name J2CAO1]
set jname [jndiName eis/j2cao1]
set j2caoAttrs [list $name $jname]
```

Using Jython:

```
ao = AdminObjectId
name = ['name', 'J2CAO1']
set jname = ['jndiName', eis/j2cao1]
j2caoAttrs = [name, jname]
```

d. If you are specifying the AdminObject attribute, also set up the following:

Using Jacl:

```
set cdattr [list adminObject $ao]
```

Using Jython:

```
cdattr = ['adminObject', ao]
```

e. Create a J2C administrative object:

Using Jacl:

```
$AdminConfig create J2CAdminObject $newra $j2caoAttrs
```

Using Jython:

```
print AdminConfig.create('J2CAdminObject', newra, j2caoAttrs)
```

Example output:

```
J2CAO1(cells/mycell/nodes/mynode|resources.xml#J2CAdminObject_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Managing the message endpoint lifecycle using scripting

Use the Jython scripting language to manage your message endpoints with the wsadmin tool. Use this topic to query your configuration for message endpoint properties, and to deactivate or reactivate a message endpoint.

About this task

The Java EE Connector Architecture (JCA) allows the application server to link inbound requests from messaging resource adapters to message endpoints. The message endpoint managed bean (MBean) is a Java Management Extensions (JMX) framework MBean that the application server associates with a message endpoint instance.

Use this topic to manage situations where messaging providers fail to deliver messages to their intended destinations. For example, a provider might fail to deliver messages to a message endpoint when its underlying Message Driven Bean attempts to commit transactions against a database server that is not responding. To troubleshoot the problem, use the wsadmin tool to temporarily disable the message endpoint from handling messages. After troubleshooting the issue, use the wsadmin tool to reactivate the message endpoint.

The steps in this topic display how to use the AdminControl object and the wsadmin tool to invoke a Message Endpoint MBean to:

- Display properties of the a message endpoint
- Temporarily deactivate a message endpoint
- Reactivate a message endpoint

Note: Starting in Version 7.0, you can use the AdminControl object and the wsadmin tool to deactivate message endpoints to pause the endpoints from receiving messages, and to reactivate message endpoints to resume message handling. Previously, the application server activated and deactivated message endpoints when the application or resource adapter was started and stopped.

- Display properties of a message endpoint.

  Use the following command to display a list of all message endpoints in your configuration:

```
AdminControl.queryNames('*:type=J2CMessageEndpoint,*')
```

  For the previous example, the command returns the following information for the `JMSMDB_MessageEndpoint`:

```
WebSphere:name=JMSMDB_J2CMessageEndpoint,
process=myServer,
platform=dynamicproxy,
cell=myNode01Cell,
node=myNode01,
version=6.1.0.0,
type=J2CMessageEndpoint,
mbeanIdentifier=cells/myNode01Cell/nodes/myNode01/servers/myServer/resources.xml#example#example.jar#JMSMDB_J2CMessageEndpoint,
spec=1.0,
Server=server1,
diagnosticProvider=false,
j2eeType=JCAMessageEndpoint,
J2EEApplication=example
J2EEServer=myServer,
J2CResourceAdapter=SIB JMS Resource Adapter,
MessageDrivenBean=example#example.jar#JMSMDBActivationSpec=3727AS1
```

- Temporarily deactivate the message endpoint.

  Use the following commands to cache the instance and deactivate the message endpoint:

```
objectName=AdminControl.queryNames('*:name=JMSMDB_MessageEndpoint,*')
AdminControl.invoke(objectName, 'pause')
```

  The command returns the following output:

```
Message Endpoint JMSMDB_J2CMessageEndpoint is deactivated
```

- Reactivate the message endpoint.

  Use the following commands to cache the instance and reactivate the message endpoint:

```
objectName=AdminControl.queryNames('*:name=JMSMDB_MessageEndpoint,*')
AdminControl.invoke(objectName, 'resume')
```

  The command returns the following output:

```
Message Endpoint JMSMDB_J2CMessageEndpoint is activated
```

Testing data source connections using scripting

You can test connections for data sources with the wsadmin tool and scripting. After you have defined and saved a data source, you can test the data source connection to ensure that the parameters in the data source definition are correct.

About this task

You can use the testConnection command for the AdminControl object to test data source connections for a cell, node, server, application, or cluster scope. This topic provides an example that tests the data source connection for the application scope.

- Test the data source connection for a cell, node, or server scope.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Identify the DataSourceCfgHelper MBean and assign it to the dshelper variable.
     - Using Jacl:

       ```
       set ds [$AdminConfig getid /DataSource:DS1/]
       $AdminControl testConnection $ds
       ```

     - Using Jython:

       ```
       ds = AdminConfig.getid('/DataSource:DS1/')
       AdminControl.testConnection(ds)
       ```

       Example output:

       ```
       WASX7217I: Connection to provided datasource was successful.
       ```

  3. Test the connection. The following example invokes the testConnectionToDataSource operation on the MBean, passing in the classname, userid, password, database name, JDBC driver class path, language, and country.
     - Using Jacl:
     - Using Jython:

       Example output:

       ```
       WASX7217I: Connection to provided data source was successful.
       ```

- Test the data source connection for an application scope.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Get the data source for the application of interest.

     Use the AdminConfig object to determine the configuration IDs of the `myApplication` application and `DSA1` data source, as the following examples demonstrate:

     - Using Jacl:

       ```
       set appID [$AdminConfig getid /Deployment:myApplication/]
       set ds [$AdminConfig list DataSource $appID]
       ```

– Using Jython:

```
appID = AdminConfig.getid("/Deployment:myApplication/")
ds = AdminConfig.list("DataSource", appID)
```

3. Test the connection.

   Use the AdminConfig object to test the connection for the data source of interest, as the following examples demonstrate:

   – Using Jacl:

   ```
   $AdminControl testConnection $ds
   ```

   – Using Jython:

   ```
   AdminControl.testConnection(ds)
   ```

   The command returns output that indicates whether the connection is successful, as demonstrated in the following sample output:

```
WASX7467I: Connection to provided datasource on node myNode processnodeagent was successful.
WASX7217I: Connection to provided datasource was successful.
```

- Test the data source connection for a cluster scope.

  In the following example, the `Cluster1` server cluster contains cluster members on the `node1`, `node2`, and `node3` nodes. The `Cluster1` server cluster contains the `DSC1` data source.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Get the data source configuration ID for the cluster of interest.

     Use the AdminConfig object to determine the configuration IDs of the `Cluster1` cluster and `DSA1` data source, as the following examples demonstrate:

     – Using Jacl:

     ```
     set cluster [$AdminConfig getid /ServerCluster:Cluster1/]
     set ds [$AdminConfig list DataSource $cluster]
     ```

     – Using Jython:

     ```
     cluster = AdminConfig.getid("/ServerCluster:Cluster1/")
     ds = AdminConfig.list("DataSource", cluster)
     ```

  3. Test the connection.

     Use the AdminConfig object to test the connection for the data source of interest, as the following examples demonstrate:

     – Using Jacl:

     ```
     $AdminControl testConnection $ds
     ```

     – Using Jython:

     ```
     AdminControl.testConnection(ds)
     ```

     The command returns output that indicates whether the connection is successful, as demonstrated in the following sample output:

```
WASX7467I: Connection to provided datasource on node node1 process nodeagent was successful.
WASX7467I: Connection to provided datasource on node node2 process nodeagent was successful.
WASX7467I: Connection to provided datasource on node node3 process nodeagent was successful.
WASX7217I: Connection to provided datasource was successful.
```

JDBCProviderManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages in interactive mode to configure data access and data sources with scripting. The commands and parameters in the JDBCProviderManagement group can be used to create or list data sources and Java database connectivity (JDBC) providers.

The JDBCProviderManagement command group for the AdminTask object includes the following commands:

- “createDatasource” on page 966
- “createJDBCProvider” on page 967

- “listDatasources” on page 968
- “listJDBCProviders” on page 969

createDatasource

The createDatasource command creates a new data source to access the back end data store. Application components use the data source to access connection instances to your database.

Target object

JDBC Provider Object ID - The configuration object of the JDBC provider to which the new data source will belong.

Required parameters

- name
The name of the data source. (String, required)

-jndiName
The Java Naming and Directory Interface (JNDI) name. (String, required)

-dataStoreHelperClassName
The name of the DataStoreHelper implementation class that extends the capabilities of the selected JDBC driver implementation class to perform data-specific functions. WebSphere Application Server provides a set of DataStoreHelper implementation classes for each of the JDBC provider drivers it supports. (String, required)

configureResourceProperties
This command step configures the resource properties that are required by the data source. (Required) You can specify the following parameters for this step:

name
The name of the resource property. (String, required)

type
The type of the resource property. (String, required)

value
The value of the resource property. (String, required)

Optional parameters

-description
The description of the data source. (String, optional)

-category
The category that you can use to classify a group of data sources. (String, optional)

-componentManagedAuthenticationAlias
The alias used for database authentication at run time. This alias is only used when the application resource reference is using res-auth=Application. (String, optional)

-containerManagedPersistence
Specifies if the data source is used for container managed persistence for enterprise beans. The default value is true. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createDatasource "DB2 Universal JDBC Driver Provider (XA)(cells/myCell|resources.xml#JDBCProvider_1180538152781)" {-name
"DB2  Universal JDBC Driver XA DataSource" -jndiName s1 -dataStoreHelperClassName com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper
 -componentManagedAuthenticationAlias myCellManager01/a1 -xaRecoveryAuthAlias myCellManager01/a1 -configureResourceProperties
{{databaseName java.lang.String db1} {driverType java.lang.Integer 4} {serverName java.lang.String  dbserver1} {portNumber java.lang.Integer 50000}}}
```

- Using Jython string:

```
AdminTask.createDatasource('DB2 Universal JDBC Driver Provider(XA) (cells/myCell|resources.xml#JDBCProvider_1180501752515)', '[-name
"DB2 Universal JDBC Driver XA DataSource 2" -jndiName ds2 -dataStoreHelperClassName com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper
 -componentManagedAuthenticationAlias myCellManager01/a1 -xaRecoveryAuthAlias myCellManager01/a1 -configureResourceProperties
[[databaseName java.lang.String db1] [driverType java.lang.Integer 4] [serverName java.lang.String  dbserver1] [portNumber java.lang.Integer 50000]]]')
```

- Using Jython list:

```
AdminTask.createDatasource('DB2 Universal JDBC Driver Provider(XA) (cells/myCell|resources.xml#JDBCProvider_1180501752515)', ['-name', '
DB2  Universal JDBC Driver XA DataSource 2', '-jndiName', 'ds2', '-dataStoreHelperClassName',
 'com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper', '-componentManagedAuthenticationAlias', 'myCellManager01/a1',
 '-xaRecoveryAuthAlias', 'myCellManager01/a1', '-configureResourceProperties', '[[databaseName java.lang.String db1]
[driverType java.lang.Integer 4] [serverName java.lang.String  dbserver1] [portNumber java.lang.Integer 50000]]']
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createDatasource {-interactive}
```

- Using Jython:

```
AdminTask.createDatasource('-interactive')
```

createJDBCProvider

The createJDBCProvider command creates a new Java database connectivity provider (JDBC) that you can use to connect to a relational database for data access.

Target object

None

Required parameters

-scope
The scope for the new JDBC provider. Provide the scope in the form `type=name`. `Type` can be Cell, Node, Server, Application, or Cluster, and `name` is the name of the specific instance of the cell, node, server, application, or cluster that you are using. The default is `none`. (String, required)

-databaseType
The type of database that will be used by the JDBC provider. For example, DB2, Derby, Informix, Oracle, and so on. (String, required)

-providerType
The JDBC provider type that will be used by the JDBC provider. (String, required)

-implementationType
The implementation type for this JDBC provider. Use `Connection pool data source` if your application runs in a single phase or a local transaction. Otherwise, use `XA data source` to run in a global transaction. (String, required)

Optional parameters

-classpath
Specifies a list of paths or JAR file names that form the location for the resource provider classes. (String, optional)

-description
The description for the JDBC provider. (String, optional)

-implementationClassName
Specifies the Java class name for the JDBC driver implementation. (String, optional)

-isolated
Specifies that the JDBC provider will load in its own class loader. The default value is false. You cannot specify a native path for an isolated JDBC provider. (Boolean, optional)

-name
The name of the JDBC provider. The default is the value from the provider template. (String, optional)

-nativePath
Specifies a list of paths that form the location for the resource provider native libraries. (String, optional)

-isolated
Specifies whether the JDBC provider loads within the class loader. The default value is `false`. You cannot specify a native path for an isolated JDBC provider. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createJDBCProvider {-scope Cell=my02Cell -databaseType DB2 -providerType "DB2 Universal JDBC Driver Provider"
-implementationType "XA data source" -name "DB2 Universal JDBC Driver Provider (XA)" -description "XA DB2 Universal JDBC Driver-compliant
Provider. Datasources created under this provider support the use of XA to perform 2-phase commit processing. Use of driver type 2 on WAS z/OS is not
supported for datasources created under this provider." -classpath
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar;${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar;${DB2UNIVERSAL_JDBC_DRIVER_PATH}/
db2jcc_license_cisuz.jar -nativePath ${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}}
```

- Using Jython string:

```
AdminTask.createJDBCProvider('[-scope Cell=myCell -databaseType DB2 -providerType "DB2 Universal JDBC Driver Provider"
 -implementationType "XA datasource" -name "DB2 Universal JDBC Driver Provider (XA)" -description "XA DB2 Universal JDBC
Driver-compliant Provider. Datasources created under this provider support the use of XA to perform 2-phase commit processing. Use of driver
 type 2 on WAS z/OS is not supported for datasources created under this provider." -classpath
 ${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar;${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar;${DB2UNIVERSAL_JDBC_DRIVER_PATH}/
db2jcc_license_cisuz.jar -nativePath ${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}]')
```

- Using Jython list:

```
AdminTask.createJDBCProvider(['-scope', 'Cell=myCell', '-databaseType', 'DB2', '-providerType', 'DB2 Universal JDBC Driver Provider',
 '-implementationType', 'XA data source', '-name', 'DB2 Universal JDBC Driver Provider (XA)', '-description', 'XA DB2 Universal
JDBC Driver-compliant Provider. Datasources created under this provider support the use of XA to perform 2-phase commit processing. Use of
driver type 2 on WAS z/OS is not supported for datasources created under this provider.', '-classpath',
  '${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar;${UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar;${DB2UNIVERSAL_JDBC_DRIVER_PATH}/
db2jcc_license_cisuz.jar', '-nativePath', '${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}']])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createJDBCProvider {-interactive}
```

- Using Jython:

```
AdminTask.createJDBCProvider('-interactive')
```

listDatasources

Use the listDatasources command to list data sources that are contained in the specified scope.

Target object

None

Required parameters

None.

Optional parameters

-scope
The scope for the data sources that will be listed. Provide the scope in the form `type=name`. `Type` can

be Cell, Node, Server, Application, or Cluster, and `name` is the name of the specific instance of the cell, node, server, application, or cluster that you are using. The default is `All`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listDatasources {-scope Cell=my02Cell}
```

- Using Jython string:

```
AdminTask.listDatasources('[-scope Cell=my02Cell]')
```

- Using Jython list:

```
AdminTask.listDatasources('-scope', 'Cell=my02Cell')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listDatasources {-interactive}
```

- Using Jython:

```
AdminTask.listDatasources('-interactive')
```

listJDBCProviders

The listJDBCProviders command lists JDBC providers that are contained in the specified scope.

Target object

None

Required parameters

None.

Optional parameters

-scope
The scope for the JDBC providers that will be listed. Provide the scope in the form `type=name`. `Type` can be Cell, Node, Server, Application, or Cluster, and `name` is the name of the specific instance of the cell, node, server, application, or cluster that you are using. The default is `All`. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listJDBCProviders {-scope Cell=my02Cell}
```

- Using Jython string:

```
AdminTask.listJDBCProviders('-scope Cell=my02Cell')
```

- Using Jython list:

```
AdminTask.listJDBCProviders(['-scope', 'Cell=my02Cell'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listJDBCProviders {-interactive}
```

- Using Jython:

```
AdminTask.listJDBCProviders('-interactive')
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 12. Configuring messaging with scripting

Use these topics to learn about configuring messaging with the wsadmin tool. You can configure the message listener service, Java Messaging Service settings, queue and connection factories, and WebSphere MQ settings.

About this task

You can use the wsadmin tool to configure various messaging connections and settings.

- Configure the message listener service.

  The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.
- Configure Java Messaging Service (JMS) providers, destinations, and connections.

  The application server supports asynchronous messaging through the use of a JMS provider and its related messaging system. You can use the wsadmin tool to configure new JMS providers, destinations, and connections.

  A JMS destination is used to configure the properties for the associated messaging provider. Connections to the JMS destination are created by the associated JMS connection factory. A JMS connection factory is used to create connections to the associated JMS provider of JMS destinations, for both point-to-point and publish/subscribe messaging.
- Configure queue and topic connection factories for the application server. Use queue and topic connection factories to create connections between providers and destinations. A queue connection factory creats a connection to the associated JMS provider of the JMS queue destinations, for point-to-point messaging. A topic connection factory creates a connection to the associated JMS provider of JMS topic destinations, for publish and subscribe messaging.
- Configure WebSphere MQ settings. A WebSphere MQ server represents either a WebSphere MQ queue manager or a WebSphere MQ queue sharing group. It is used by Service Integration Bus messaging to define properties used for connecting to WebSphere MQ. Setting up a WebSphere MQ server involves using the administrative console to create the server definition, add it to a service integration bus, and create a WebSphere MQ queue type destination.

Configuring the message listener service using scripting

Use scripting to configure the message listener service.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure the message listener service for an application server:

1. Identify the application server and assign it to the server variable:
   - Using Jacl:

     ```
     set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```
   - Using Jython:

     ```
     server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print server
     ```

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

2. Identify the message listener service belonging to the server and assign it to the mls variable:
   - Using Jacl:

     ```
     set mls [$AdminConfig list MessageListenerService $server]
     ```

   - Using Jython:

     ```
     mls = AdminConfig.list('MessageListenerService', server)
     print mls
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#MessageListenerService_1)
   ```

3. Modify various attributes with one of the following examples:
   - This example command changes the thread pool attributes:
     - Using Jacl:

       ```
       $AdminConfig modify $mls {{threadPool {{inactivityTimeout 4000}
       {isGrowable true} {maximumSize 100} {minimumSize 25}}}}
       ```

     - Using Jython:

       ```
       AdminConfig.modify(mls, [['threadPool', [['inactivityTimeout', 4000],
       ['isGrowable',  'true'], ['maximumSize', 100], ['minimumSize', 25]]]])
       ```

   - This example modifies the property of the first listener port:
     - Using Jacl:

       ```
       set lports [$AdminConfig showAttribute $mls listenerPorts]
       set lport [lindex $lports 0]
       $AdminConfig modify $lport {{maxRetries 2}}
       ```

     - Using Jython:

       ```
       lports = AdminConfig.showAttribute(mls, 'listenerPorts')
       cleanLports = lports[1:len(lports)-1]
       lport = cleanLports.split(" ")[0]
       AdminConfig.modify(lport, [['maxRetries', 2]])
       ```

   - This example adds a listener port:
     - Using Jacl:

       ```
       set new [$AdminConfig create ListenerPort $mls {{name my}
       {destinationJNDIName di} {connectionFactoryJNDIName jndi/fs}}]
       $AdminConfig create StateManageable $new {{initialState START}}
       ```

     - Using Jython:

       ```
       new = AdminConfig.create('ListenerPort', mls, [['name', 'my'],
       ['destinationJNDIName', 'di'], ['connectionFactoryJNDIName', 'jndi/fsi']])
       print new
       print AdminConfig.create('StateManageable', new, [['initialState', 'START']])
       ```

     Example output:

     ```
     my(cells/mycell/nodes/mynode/servers/server1:server.xml#ListenerPort_1079471940692)
     (cells/mycell/nodes/mynode/servers/server1:server.xml#StateManageable_107947182623)
     ```

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new JMS providers using scripting

You can use the wsadmin tool and scripting to configure a new JMS provider.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new JMS provider:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
     ```
   - Using Jython:

     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```
2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required JMSProvider
     ```
   - Using Jython:

     ```
     print AdminConfig.required('JMSProvider')
     ```

   Example output:

   ```
   Attribute        Type
   name     String
   externalInitialContextFactory    String
   externalProviderURL               String
   ```
3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name JMSP1]
     set extICF [list externalInitialContextFactory
     "Put the external initial context factory here"]
     set extPURL [list externalProviderURL "Put the external provider URL here"]
     set jmspAttrs [list $name $extICF $extPURL]
     ```
   - Using Jython:

     ```
     name = ['name', 'JMSP1']
     extICF = ['externalInitialContextFactory',
     "Put the external initial context factory here"]
     extPURL = ['externalProviderURL', "Put the external provider URL here"]
     jmspAttrs = [name, extICF, extPURL]
     print jmspAttrs
     ```

   Example output:

   ```
   {name JMSP1} {externalInitialContextFactory {Put the external
   initial context factory here }} {externalProviderURL
   {Put the external provider URL here}}
   ```
4. Create the JMS provider:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig create JMSProvider $node $jmspAttrs]
     ```
   - Using Jython:

     ```
     newjmsp = AdminConfig.create('JMSProvider', node, jmspAttrs)
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new JMS destinations using scripting

You can use scripting and the wsadmin tool to configure a new JMS destination.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new JMS destination:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
     ```
   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required GenericJMSDestination
     ```
   - Using Jython:

     ```
     print AdminConfig.required('GenericJMSDestination')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   externalJNDIName  String
   ```
3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name JMSD1]
     set jndi [list jndiName jms/JMSDestination1]
     set extJndi [list externalJNDIName jms/extJMSD1]
     set jmsdAttrs [list $name $jndi $extJndi]
     ```
   - Using Jython:

     ```
     name = ['name', 'JMSD1']
     jndi = ['jndiName', 'jms/JMSDestination1']
     extJndi = ['externalJNDIName', 'jms/extJMSD1']
     jmsdAttrs = [name, jndi, extJndi]
     print jmsdAttrs
     ```

   Example output:

   ```
   {name JMSD1} {jndiName jms/JMSDestination1} {externalJNDIName jms/extJMSD1}
   ```
4. Create generic JMS destination:
   - Using Jacl:

     ```
     $AdminConfig create GenericJMSDestination $newjmsp  $jmsdAttrs
     ```

- Using Jython:

```
print AdminConfig.create('GenericJMSDestination', newjmsp,  jmsdAttrs)
```

Example output:

```
JMSD1(cells/mycell/nodes/mynode|resources.xml#GenericJMSDestination_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new JMS connections using scripting

Use scripting and the wsadmin tool to configure a new JMS connection.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new JMS connection:

1. Configure a connection pool for your generic connection factories.

   Because Java 2 Connector (J2C) manages the generic connection factories, you must configure a connection pool to indicate the policy for connection management by J2C. The following example commands configure a connection pool in your environment:

   - Using Jacl:

```
set connectionPool [$AdminConfig create ConnectionPool $yourGenericCF {} connectionPool]
set sessionPool [$AdminConfig create ConnectionPool $yourGenericCF {} sessionPool]
```

   - Using Jython:

```
connectionPool = AdminConfig.create('ConnectionPool', '[yourGenericCF {}, connectionPool]')
sessionPool = AdminConfig.create('ConnectionPool', '[yourGenericCF {}, sessionPool]')
```

2. Identify the parent ID:

   - Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
```

   - Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

   Example output:

```
JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
```

3. Get required attributes:

   - Using Jacl:

```
$AdminConfig required GenericJMSConnectionFactory
```

   - Using Jython:

```
print AdminConfig.required('GenericJMSConnectionFactory')
```

   Example output:

```
Attribute       Type
name       String
jndiName    String
externalJNDIName  String
```

4. Set up required attributes:

   - Using Jacl:

```
set name [list name JMSCF1]
set jndi [list jndiName jms/JMSConnFact1]
set extJndi [list externalJNDIName jms/extJMSCF1]
set jmscfAttrs [list $name $jndi $extJndi]
```

Example output:

```
{name JMSCF1} {jndiName jms/JMSConnFact1} {externalJNDIName jms/extJMSCF1}
```

- Using Jython:

```
name = ['name', 'JMSCF1']
jndi = ['jndiName', 'jms/JMSConnFact1']
extJndi = ['externalJNDIName', 'jms/extJMSCF1']
jmscfAttrs = [name, jndi, extJndi]
print jmscfAttrs
```

Example output:

```
[[name, JMSCF1], [jndiName, jms/JMSConnFact1], [externalJNDIName, jms/extJMSCF1]]
```

5. Create generic JMS connection factory:
   - Using Jacl:

     ```
     $AdminConfig create GenericJMSConnectionFactory $newjmsp $jmscfAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('GenericJMSConnectionFactory', newjmsp, jmscfAttrs)
     ```

   Example output:

   ```
   JMSCF1(cells/mycell/nodes/mynode|resources.xml#GenericJMSConnectionFactory_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WebSphere queue connection factories using scripting

You can use scripting and the wsadmin tool to configure new queue connection factories in WebSphere Application Server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WebSphere queue connection factory:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set v5jmsp [$AdminConfig getid "/Cell:mycell/Node:mynode/JMSProvider:WebSphere JMS Provider/"]
     ```

   - Using Jython:

     ```
     v5jmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:WebSphere JMS Provider/')
     print v5jmsp
     ```

   Example output:

   ```
   "WebSphere JMS Provider(cells/mycell/nodes/mynode|resources.xml#builtin_jmsprovider)"
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASQueueConnectionFactory
     ```

- Using Jython:

```
print AdminConfig.required('WASQueueConnectionFactory')
```

Example output:

```
Attribute       Type
name       String
jndiName    String
```

3. Set up required attributes:
   - Using Jacl:

```
set name [list name WASQCF]
set jndi [list jndiName jms/WASQCF]
set mqcfAttrs [list $name $jndi]
```

Example output:

```
{name WASQCF} {jndiName jms/WASQCF}
```

   - Using Jython:

```
name = ['name', 'WASQCF']
jndi = ['jndiName', 'jms/WASQCF']
mqcfAttrs = [name, jndi]
print mqcfAttrs
```

Example output:

```
[[name, WASQCF], [jndiName, jms/WASQCF]]
```

4. Set up a template:
   - Using Jacl:

```
set template [lindex [$AdminConfig listTemplates WASQueueConnectionFactory] 0]
```

   - Using Jython:

```
lineseparator = java.lang.System.getProperty('line.separator')
template = AdminConfig.listTemplates('WASQueueConnectionFactory').split(lineseparator)[0]
print template
```

5. Create was queue connection factories:
   - Using Jacl:

```
$AdminConfig createUsingTemplate WASQueueConnectionFactory $v5jmsp $mqcfAttrs $template
```

   - Using Jython:

```
AdminConfig.createUsingTemplate('WASQueueConnectionFactory', v5jmsp, mqqcfAttrs, template)
```

Example output:

```
WASQCF(cells/mycell/nodes/mynode|resources.xml#WASQueueConnectionFactory_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WebSphere topic connection factories using scripting

Use scripting and the wsadmin tool to configure new WebSphere topic connection factories.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WebSphere topic connection factory:

1. Identify the parent ID:

- Using Jacl:

```
set v5jmsp [$AdminConfig getid "/Cell:mycell/Node:mynode/JMSProvider:WebSphere JMS Provider/"]
```

- Using Jython:

```
v5jmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:WebSphere JMS Provider/')
print v5jmsp
```

Example output:

```
"WebSphere JMS Provider(cells/mycell/nodes/mynode|resources.xml#builtin_jmsprovider)"
```

2. Get required attributes:
   - Using Jacl:

```
$AdminConfig required WASTopicConnectionFactory
```

   - Using Jython:

```
print AdminConfig.required('WASTopicConnectionFactory')
```

   Example output:

```
Attribute            Type
name                 String
jndiName             String
port                 ENUM(DIRECT, QUEUED)
```

3. Set up required attributes:
   - Using Jacl:

```
set name [list name WASTCF]
set jndi [list jndiName jms/WASTCF]
set port [list port QUEUED]
set mtcfAttrs [list $name $jndi $port]
```

     Example output:

```
{name WASTCF} {jndiName jms/WASTCF} {port QUEUED}
```

   - Using Jython:

```
name = ['name', 'WASTCF']
jndi = ['jndiName', 'jms/WASTCF']
port = ['port', 'QUEUED']
mtcfAttrs = [name, jndi, port]
print mtcfAttrs
```

     Example output:

```
[[name, WASTCF], [jndiName, jms/WASTCF], [port, QUEUED]]
```

4. Create was topic connection factories:
   - Using Jacl:

```
$AdminConfig create WASTopicConnectionFactory $v5jmsp $mtcfAttrs
```

   - Using Jython:

```
print AdminConfig.create('WASTopicConnectionFactory', v5jmsp, mtcfAttrs)
```

   Example output:

```
WASTCF(cells/mycell/nodes/mynode|resources.xml#WASTopicConnectionFactory_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WebSphere queues using scripting

You can use scripting to configure a new WebSphere queue.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WebSphere queue:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set v5jmsp [$AdminConfig getid "/Cell:mycell/Node:mynode/JMSProvider:WebSphere JMS Provider/"]
     ```
   - Using Jython:

     ```
     v5jmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:WebSphere JMS Provider/')
     print v5jmsp
     ```

   Example output:

   ```
   "WebSphere JMS Provider(cells/mycell/nodes/mynode|resources.xml#builtin_jmsprovider)"
   ```
2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASQueue
     ```
   - Using Jython:

     ```
     print AdminConfig.required('WASQueue')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```
3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WASQ1]
     set jndi [list jndiName jms/WASQ1]
     set wqAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name WASQ1} {jndiName jms/WASQ1}
     ```
   - Using Jython:

     ```
     name = ['name', 'WASQ1']
     jndi = ['jndiName', 'jms/WASQ1']
     wqAttrs = [name, jndi]
     print wqAttrs
     ```

     Example output:

     ```
     [[name, WASQ1], [jndiName, jms/WASQ1]]
     ```
4. Create was queue:
   - Using Jacl:

     ```
     $AdminConfig create WASQueue $v5jmsp $wqAttrs
     ```
   - Using Jython:

     ```
     print AdminConfig.create('WASQueue', v5jmsp, wqAttrs)
     ```

   Example output:

   ```
   WASQ1(cells/mycell/nodes/mynode|resources.xml#WASQueue_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new WebSphere topics using scripting

You can configure new WebSphere topics using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new WebSphere topic:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set v5jmsp [$AdminConfig getid "/Cell:mycell/Node:mynode/JMSProvider:WebSphere JMS Provider/"]
     ```
   - Using Jython:

     ```
     v5jmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:WebSphere JMS Provider/')
     print v5jmsp
     ```

   Example output:

   ```
   "WebSphere JMS Provider(cells/mycell/nodes/mynode|resources.xml#builtin_jmsprovider)"
   ```
2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASTopic
     ```
   - Using Jython:

     ```
     print AdminConfig.required('WASTopic')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   topic        String
   ```
3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WAST1]
     set jndi [list jndiName jms/WAST1]
     set topic [list topic "Put your topic here"]
     set wtAttrs [list $name $jndi $topic]
     ```

     Example output:

     ```
     {name WAST1} {jndiName jms/WAST1} {topic {Put your topic here}}
     ```
   - Using Jython:

     ```
     name = ['name', 'WAST1']
     jndi = ['jndiName', 'jms/WAST1']
     topic = ['topic', "Put your topic here"]
     wtAttrs = [name, jndi, topic]
     print wtAttrs
     ```

     Example output:

     ```
     [[name, WAST1], [jndiName, jms/WAST1], [topic, "Put your topic here"]]
     ```
4. Create was topic:
   - Using Jacl:

     ```
     $AdminConfig create WASTopic $v5jmsp $wtAttrs
     ```
   - Using Jython:

     ```
     print AdminConfig.create('WASTopic', v5jmsp, wtAttrs)
     ```

Example output:

```
WAST1(cells/mycell/nodes/mynode|resources.xml#WASTopic_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring a new connection factory for the WebSphere MQ messaging provider using scripting

You can use scripting to configure a new connection factory for the WebSphere MQ messaging provider.

Before you begin

You can also use the createWMQConnectionFactory command to create a connection factory for the WebSphere MQ messaging provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a connection factory for the WebSphere MQ messaging provider:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/
     JMSProvider:WebSphere MQ JMS Provider/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/
     JMSProvider:WebSphere MQ JMS Provider')
     print newjmsp
     ```

   Example output:

   ```
   WebSphere MQ JMS Provider(cells/mycell/nodes/mynode|
   resources.xml#builtin_mqprovider)
   ```

2. Get the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQConnectionFactory')
     ```

   Example output:

   ```
   attribute    Type
   name         String
   jndiName     String
   ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name MQCF]
     set jndi [list jndiName jms/MQCF]
     set mqcfAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name MQCF} {jndiName jms/MQCF}
     ```

   - Using Jython:

```
name = ['name', 'MQCF']
jndi = ['jndiName', 'jms/MQCF']
mqcfAttrs = [name, jndi]
print mqcfAttrs
```

Example output:

```
[[name, MQCF], [jndiName, jms/MQCF]]
```

4. Set up a template:
   - Using Jacl:

     ```
     set template [lindex [$AdminConfig listTemplates MQConnectionFactory] 0]
     ```

   - Using Jython:

     ```
     import java
     lineseparator = java.lang.System.getProperty('line.separator')
     template =
     AdminConfig.listTemplates('MQConnectionFactory').split(lineseparator)[0]
     print template
     ```

   Example output:

   ```
   Example non-XA WMQ ConnectionFactory(templates/
   system:JMS-resource-provider-templates.xml
   #MQConnectionFactory_3)
   ```

5. Create a connection factory for the WebSphere MQ messaging provider:
   - Using Jacl:

     ```
     $AdminConfig createUsingTemplate MQConnectionFactory
     $newjmsp $mqcfAttrs $template
     ```

   - Using Jython:

     ```
     print AdminConfig.createUsingTemplate('MQConnectionFactory',
     newjmsp, mqcfAttrs, template)
     ```

   Example output:

   ```
   MQCF(cells/mycell/nodes/mynode:resources.xml#MQConnectionFactory_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Defining a connection factory for the WebSphere MQ messaging provider by supplying custom connection data

The following example creates a connection factory, specifying custom connection data. Due to the default values assumed for the unspecified parameters, applications using this connection factory expect to be co-located with a queue manager installed on the same node.

```
wsadmin>$AdminConfig getid /Node:9994GKCNode01
9994GKCNode01(cells/9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)
wsadmin>$AdminTask createWMQConnectionFactory
9994GKCNode01(cells/9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)
{-name cf1 –jndiName "jms/cf/cf1" –type CF}
cf1(cells/9994GKCNode01Cell/nodes/9994GKCNode01|resources.xml#
MQConnectionFactory_1205322636000)
```

The following example creates an connection factory for which the user must specify and maintain all the parameters used for establishing a connection to WebSphere MQ.

- Using Jython:

  ```
  wsadmin>AdminConfig.getid("/Node:9994GKCNode01")
  9994GKCNode01(cells/9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)

  wsadmin>AdminTask.createWMQConnectionFactory("9994GKCNode01(cells/
  9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)", ["-name cf2
  ```

```
 –jndiName 'jms/cf/cf2' –type CF –description 'Must remember to keep each
of these connection factories in sync with the WebSphere MQ queue manager
to which they refer' –qmgrName QM1 –qmgrHostname 192.168.0.22 –qmgrPort 1415
 –qmgrSvrconnChannel QM1.SVRCONN"])
cf2(cells/9994GKCNode01Cell/nodes/9994GKCNode01|resources.xml#
MQConnectionFactory_120532263601)
```

- Using Jacl:

```
wsadmin>$AdminConfig getid /Node:9994GKCNode01
9994GKCNode01(cells/9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)

wsadmin>$AdminTask createWMQConnectionFactory
9994GKCNode01(cells/9994GKCNode01Cell/nodes/9994GKCNode01|node.xml#Node_1)
{-name cf2 –jndiName "jms/cf/cf2" –type CF –description "Must remember to
keep each of these connection factories in sync with the WebSphere MQ queue
manager to which they refer" –qmgrName QM1 –qmgrHostname 192.168.0.22
–qmgrPort 1415 –qmgrSvrconnChannel QM1.SVRCONN}
cf2(cells/9994GKCNode01Cell/nodes/9994GKCNode01|resources.xml#
MQConnectionFactory_120532263601)
```

Configuring a new queue connection factory for the WebSphere MQ messaging provider using scripting

You can use scripting to configure a new queue connection factory for the WebSphere MQ messaging provider.

Before you begin

You can also use the createWMQConnectionFactory command to create a queue connection factory for the WebSphere MQ messaging provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new queue connection factory for the WebSphere MQ messaging provider:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQQueueConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQQueueConnectionFactory')
     ```

   Example output:

   ```
   attribute  Type
   name       String
   jndiName   String
   ```

3. Set up the required attributes:

- Using Jacl:

```
set name [list name MQQCF]
set jndi [list jndiName jms/MQQCF]
set mqqcfAttrs [list $name $jndi]
```

  Example output:

```
{name MQQCF} {jndiName jms/MQQCF}
```

- Using Jython:

```
name = ['name', 'MQQCF']
jndi = ['jndiName', 'jms/MQQCF']
mqqcfAttrs = [name, jndi]
print mqqcfAttrs
```

  Example output:

```
[[name, MQQCF], [jndiName, jms/MQQCF]]
```

4. Set up a template:
   - Using Jacl:

```
set template [lindex [$AdminConfig listTemplates MQQueueConnectionFactory] 0]
```

   - Using Jython:

```
import java
lineseparator = java.lang.System.getProperty('line.separator')
template = AdminConfig.listTemplates('MQQueueConnectionFactory').
split(lineseparator)[0]
print template
```

   Example output:

```
Example non-XA WMQ QueueConnectionFactory(templates/
system:JMS-resource-provider-templates.xml
#MQQueueConnectionFactory_3)
```

5. Create a queue connection factory for the WebSphere MQ messaging provider:
   - Using Jacl:

```
$AdminConfig createUsingTemplate MQQueueConnectionFactory
$newjmsp $mqqcfAttrs $template
```

   - Using Jython:

```
print AdminConfig.createUsingTemplate('MQQueueConnectionFactory',
newjmsp, mqqcfAttrs, template)
```

   Example output:

```
MQQCF(cells/mycell/nodes/mynode:resources.xml#MQQueueConnectionFactory_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Defining a queue connection factory for the WebSphere MQ messaging provider by supplying custom connection data

The following example creates a queue connection factory, specifying custom connection data.

```
wsadmin>AdminTask.createWMQConnectionFactory('"WebSphere MQ JMS Provider
(cells/EXAMPLECell01|resources.xml#builtin_mqprovider)"', '
[-type QCF -name MQQueueConnectionFactory1 -jndiName
jms/MQQueueConnectionFactory1 -description -qmgrName
QueueManagerName -wmqTransportType BINDINGS_THEN_CLIENT
-qmgrHostname HostName -qmgrSvrconnChannel ServerConnectionChannel ]')
```

The following example creates a queue connection factory, specifying CCDT connection data.

```
wsadmin>AdminTask.createWMQConnectionFactory('"WebSphere MQ JMS Provider
(cells/EXAMPLECell01|resources.xml#builtin_mqprovider)"', '
[-type QCF -name MQQueueConnectionFactory2 -jndiName
jms/MQQueueConnectionFactory2 -description -ccdtUrl
http://ClientChannelDefinitionTableURL -ccdtQmgrName QueueManager ]')
```

Configuring a new topic connection factor for the WebSphere MQ messaging provider using scripting

You can use scripting to configure a new topic connection factory for the WebSphere MQ messaging provider.

Before you begin

You can also use the createWMQConnectionFactory command to create a topic connection factory for the WebSphere MQ messaging provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a topic connection factory for the WebSphere MQ messaging provider:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/
     JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/
     JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
   ```

2. Get the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQTopicConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQTopicConnectionFactory')
     ```

   Example output:

   ```
   attribute    Type
   name      String
   jndiName   String
   ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name MQTCF]
     set jndi [list jndiName jms/MQTCF]
     set mqtcfAttrs [list $name  $jndi]
     ```

     Example output:

     ```
     {name MQTCF} {jndiName jms/MQTCF}
     ```

   - Using Jython:

```
name = ['name', 'MQTCF']
jndi = ['jndiName', 'jms/MQTCF']
mqtcfAttrs = [name, jndi]
print  mqtcfAttrs
```

Example output:

```
[[name, MQTCF], [jndiName, jms/MQTCF]]
```

4. Set up a template:
   - Using Jacl:

     ```
     set template [lindex [$AdminConfig listTemplates MQTopicConnectionFactory] 0]
     ```

   - Using Jython:

     ```
     import java
     lineseparator = java.lang.System.getProperty('line.separator')
     template = AdminConfig.listTemplates('MQTopicConnectionFactory').
     split(lineseparator)[0]
     print template
     ```

   Example output:

   ```
   Example non-XA WMQ TopicConnectionFactory(templates/
   system: JMS-resource-provider-templates.xml
   #MQTopicConnectionFactory_5)
   ```

5. Create a topic connection factory for the WebSphere MQ messaging provider:
   - Using Jacl:

     ```
     $AdminConfig createUsingTemplate MQTopicConnectionFactory
     $mqjmsp $mqtcfAttrs $template
     ```

   - Using Jython:

     ```
     print  AdminConfig.createUsingTemplate('MQTopicConnectionFactory',
     mqjmsp, mqtcfAttrs, template)
     ```

   Example output:

   ```
   MQTCF(cells/mycell/nodes/mynode:resources.xml#MQTopicConnectionFactory_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Defining a topic connection factory for the WebSphere MQ messaging provider by supplying custom connection data

The following example creates a topic connection factory, specifying custom connection data.

Using Jython:

```
wsadmin>AdminTask.createWMQConnectionFactory('"WebSphere MQ JMS Provider
(cells/EXAMPLECell01|resources.xml#builtin_mqprovider)"', '
[-type TCF -name MQTopicConnectionFactory1 -jndiName
 jms/MQTopicConnectionFactory1 -description -qmgrName
QueueManagerName -wmqTransportType BINDINGS_THEN_CLIENT
-qmgrHostname HostName -qmgrSvrconnChannel ServerConnectionChannel ]')
```

The following example creates a topic connection factory, specifying CCDT connection data.

Using Jython:

```
wsadmin>AdminTask.createWMQConnectionFactory('"WebSphere MQ JMS Provider
(cells/EXAMPLECell01|resources.xml#builtin_mqprovider)"', '
[-type TCF -name MQTopicConnectionFactory2 -jndiName
jms/MQTopicConnectionFactory2 -description -ccdtUrl
http://ClientChannelDefinitionTableURL -ccdtQmgrName QueueManager ]')
```

Configuring a new queue for the WebSphere MQ messaging provider using scripting

You can use scripting to configure a new queue for the WebSphere MQ messaging provider.

Before you begin

You can also use the createWMQQueue command to create a queue for the WebSphere MQ messaging provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new queue for the WebSphere MQ messaging provider:

1. Identify the parent ID:
   - Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

   - Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1') print newjmsp
```

   Example output:

```
JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
```

2. Get the required attributes:
   - Using Jacl:

```
$AdminConfig required MQQueue
```

   - Using Jython:

```
print AdminConfig.required('MQQueue')
```

   Example output:

```
Attribute       Type name       String jndiName    String baseQueueName  String
```

3. Set up the required attributes:
   - Using Jacl:

```
set name [list name MQQ] set jndi [list jndiName jms/MQQ] set baseQN [list baseQueueName "Put the
 base queue name here"] set mqqAttrs [list $name $jndi $baseQN]
```

   Example output:

```
{name MQQ} {jndiName jms/MQQ} {baseQueueName {Put the base queue name here}}
```

   - Using Jython:

```
name = ['name', 'MQQ'] jndi = ['jndiName', 'jms/MQQ'] baseQN = ['baseQueueName', "Put the base
 queue name here"] mqqAttrs = [name, jndi, baseQN] print mqqAttrs
```

   Example output:

```
[[name, MQQ], [jndiName, jms/MQQ], [baseQueueName, "Put the base queue name here"]]
```

4. Set up a template:
   - Using Jacl:

```
set template [lindex [$AdminConfig listTemplates MQQueue] 0]
```

   - Using Jython:

```
import java lineseparator = java.lang.System.getProperty('line.separator') template =
 AdminConfig.listTemplates('MQQueue').split(lineseparator)[0] print template
```

   Example output:

```
Example.JMS.WMQ.Q1(templates/system:JMS-resource-provider- templates.xml#MQQueue_1)
```

5. Create a queue for the WebSphere MQ messaging provider:
   - Using Jacl:

```
$AdminConfig createUsingTemplate MQQueue $newjmsp $mqqAttrs $template
```

- Using Jython:

```
print AdminConfig.createUsingTemplate('MQQueue', newjmsp, mqqAttrs, template)
```

Example output:

```
MQQ(cells/mycell/nodes/mynode|resources.xml#MQQueue_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Defining a queue for the WebSphere MQ messaging provider

```
AdminTask.createWMQQueue('EXAMPLECell01(cells/EXAMPLECell01|cell.xml)', '[-name MQQueue
 -jndiName jms/MQQueue -queueName QueueName -qmgr QueueManager -description ]')
```

Configuring a new topic for the WebSphere MQ messaging provider using scripting

You can use scripting to configure a new topic for the WebSphere MQ messaging provider.

Before you begin

You can also use the createWMQTopic command to create a queue for the WebSphere MQ messaging provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new topic for the WebSphere MQ messaging provider:

1. Identify the parent ID:
   - Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

   - Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1') print newjmsp
```

   Example output:

```
JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
```

2. Get the required attributes:
   - Using Jacl:

```
$AdminConfig required MQTopic
```

   - Using Jython:

```
print AdminConfig.required('MQTopic')
```

   Example output:

```
Attribute            Type name                 String jndiName             String baseTopicName
      String
```

3. Set up the required attributes:
   - Using Jacl:

```
set name [list name MQT] set jndi [list jndiName jms/MQT] set baseTN [list baseTopicName "Put the
 base topic name here"] set mqtAttrs [list $name $jndi $baseTN]
```

   Example output:

```
{name MQT} {jndiName jms/MQT} {baseTopicName {Put the base topic name here}}
```

   - Using Jython:

```
name = ['name', 'MQT'] jndi = ['jndiName', 'jms/MQT'] baseTN = ['baseTopicName', "Put the base
 topic name here"] mqtAttrs = [name, jndi, baseTN] print mqtAttrs
```

Example output:

```
[[name, MQT], [jndiName, jms/MQT], [baseTopicName, "Put the base topic name here"]]
```

4. Create a topic for the WebSphere MQ messaging provider:
   - Using Jacl:

```
$AdminConfig create MQTopic $newjmsp $mqtAttrs
```

   - Using Jython:

```
print AdminConfig.create('MQTopic', newjmsp, mqtAttrs)
```

   Example output:

```
MQT(cells/mycell/nodes/mynode|resources.xml#MQTopic_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Defining a topic for the WebSphere MQ messaging provider

```
AdminTask.createWMQTopic('DRAPERDCell01(cells/DRAPERDCell01|cell.xml)', '[-name MQQueue
 -jndiName jms/MQQueue -topicName TopicName -qmgr QueueManager -description ]')
```

JCAManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure messaging with scripting. The commands and parameters in the JCA management group can be used to create and manage resource adapters, Java EE Connector Architecture (J2C) activation specifications, administrative objects, connection factories, administrative object interfaces, and message listener types.

The JCAManagement command group for the AdminTask object includes the following commands:

- “copyResourceAdapter”
- “createJ2CActivationSpec” on page 990
- “createJ2CAdminObject” on page 991
- “createJ2CConnectionFactory” on page 992
- “listAdminObjectInterfaces” on page 992
- “listConnectionFactoryInterfaces” on page 993
- “listJ2CActivationSpecs” on page 993
- “listJ2CAdminObjects” on page 994
- “listJ2CConnectionFactories” on page 995
- “listMessageListenerTypes” on page 995

copyResourceAdapter

Use the copyResourceAdapter command to create a Java 2 Connector (J2C) resource adapter under the scope that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

-name
Indicates the name of the new J2C resource adapter. This parameter is required.

-scope
Indicates the scope object ID. This parameter is required.

-useDeepCopy
If you set this parameter to `true`, all of the J2C connection factory, J2C activation specification, and J2C administrative objects will be copied to the new J2C resource adapter (deep copy). If you set this parameter to `false`, the objects are not created (shallow copy). The default is `false`.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask copyResourceAdapter $ra [subst {-name newRA -scope $scope}]
```

- Using Jython string:

```
AdminTask.copyResourceAdapter(ra, '[-name newRA -scope scope]')
```

- Using Jython list:

```
AdminTask.copyResourceAdapter(ra, ['-name', 'newRA', '-scope', 'scope'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask copyResourceAdapter {-interactive}
```

- Using Jython:

```
AdminTask.copyResourceAdapter('-interactive')
```

createJ2CActivationSpec

Use the createJ2CActivationSpec command to create a Java 2 Connector (J2C) activation specification under a J2C resource adapter and the attributes that you specify. Use the messageListenerType parameter to indicate the activation specification that is defined for the J2C resource adapter.

Target object

J2C Resource Adapter object ID

Parameters and return values

- messageListenerType
Identifies the activation specification for the J2C activation specification to be created. Use this parameter to identify the activation specification template for the J2C resource adapter that you specify.

-name
Indicates the name of the J2C activation specification that you are creating.

-jndiName
Indicates the name of the Java Naming and Directory Interface (JNDI).

-destinationJndiName
Indicates the name of the Java Naming and Directory Interface (JNDI) of corresponding destination.

-authenticationAlias
Indicates the authentication alias of the J2C activation specification that you are creating.

-description
Description of the created J2C activation specification.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createJ2CActivationSpec $ra {-name J2CAct Spec -jndiName eis/ActSpec1
 -messageListenerType javax.jms.MessageListener }
```

- Using Jython string:

```
AdminTask.createJ2CActivationSpec(ra, '[-name J2CActSpec -jndiName eis/ActSpec1
 -messageListenerType javax.jms.MessageListener]')
```

- Using Jython list:

```
AdminTask.createJ2CActivationSpec(ra, ['-name', 'J2CActSpec', '-jndiName', 'eis/ActSpec1',
 '-messageListenerType', 'javax.jms.MessageListener'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createJ2CActivationSpec {-interactive}
```

- Using Jython:

```
AdminTask.createJ2CActivationSpec('-interactive')
```

createJ2CAdminObject

Use the createJ2CAdminObject command to create an administrative object under a resource adapter with attributes that you specify. Use the administrative object interface to indicate the administrative object that is defined in the resource adapter.

Target object

J2C Resource Adapter object ID

Parameters and return values

-adminObjectInterface
Specifies the administrative object interface to identify the administrative object for the resource adapter that you specify. This parameter is required.

-name
Indicates the name of the administrative object.

-jndiName
Specifies the name of the Java Naming and Directory Interface (JNDI).

-description
Description of the created J2C admin object.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createJ2CAdminObject $ra {-adminObjectInterface fvt.adapter.message.FVTMessageProvider
 -name J2CA01 -jndiName eis/J2CA01}
```

- Using Jython string:

```
AdminTask.createJ2CAdminObject(ra, '[-adminObjectInterface fvt.adapter.message.FVTMessageProvider
 -name J2CA01 -jndiName eis/J2CA01]')
```

- Using Jython list:

```
AdminTask.createJ2CAdminObject(ra, ['-adminObjectInterface',
 'fvt.adapter.message.FVTMessageProvider', '-name', 'J2CA01', '-jndiName', 'eis/J2CA01'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createJ2CAdminObject {-interactive}
```

- Using Jython:

```
AdminTask.createJ2CAdminObject('-interactive')
```

createJ2CConnectionFactory

Use the createJ2CConnectionFactory command to create a Java 2 connection factory under a Java 2 resource adapter and the attributes that you specify. Use the connection factory interfaces to indicate the connection definitions that are defined for the Java 2 resource adapter.

Target object

J2C Resource Adapter object ID

Parameters and return values

-connectionFactoryInterface
Identifies the connection definition for the Java 2 resource adapter that you specify. This parameter is required.

-name
Indicates the name of the connection factory.

-jndiName
Indicates the name of the Java Naming and Directory Interface (JNDI).

-description
Description of the created J2C connection factory.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createJ2CConnectionFactory $ra {-connectionFactoryInterfaces javax.sql.DataSource
-name J2CCF1 -jndiName eis/J2CCF1}
```

- Using Jython string:

```
AdminTask.createJ2CConnectionFactory(ra, '[-connectionFactoryInterfaces javax.sql.DataSource
-name J2CCF1 -jndi Name eis/J2CCF1]')
```

- Using Jython list:

```
AdminTask.createJ2CConnectionFactory(ra, ['-connectionFactoryInterfaces',
 'javax.sql.DataSource', '-name', 'J2CCF1', '-jndiName', 'eis/J2CCF1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createJ2CConnectionFactory {-interactive}
```

- Using Jython:

```
AdminTask.createJ2CConnectionFactory('-interactive')
```

listAdminObjectInterfaces

Use the listAdminObjectInterfaces command to list the administrative object interfaces that are defined under the resource adapter that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

- Parameters: None

- Returns: A list of administrative object interfaces.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listAdminObjectInterfaces $ra
```

- Using Jython string:

```
AdminTask.listAdminObjectInterfaces(ra)
```

- Using Jython list:

```
AdminTask.listAdminObjectInterfaces(ra)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listAdminObjectInterfaces {-interactive}
```

- Using Jython:

```
AdminTask.listAdminObjectInterfaces('-interactive')
```

listConnectionFactoryInterfaces

Use the listConnectionFactoryInterfaces command to list all of the connection factory interfaces that are defined under the Java 2 connector resource adapter that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

- Parameters: None
- Returns: A list of connection factory interfaces.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listConnectionFactoryInterfaces $ra
```

- Using Jython string:

```
AdminTask.listConnectionFactoryInterfaces(ra)
```

- Using Jython list:

```
AdminTask.listConnectionFactoryInterfaces(ra)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listConnectionFactoryInterfaces {-interactive}
```

- Using Jython:

```
AdminTask.listConnectionFactoryInterfaces('-interactive')
```

listJ2CActivationSpecs

Use the listJ2CActivationSpecs command to list the activation specifications that are contained under the resource adapter and message listener type that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

-messageListenerType
Specifies the message listener type for the resource adapter for which you are making a list. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listJ2CActivationSpecs $ra {-messageListener Type javax.jms.MessageListener}
```

- Using Jython string:

```
AdminTask.listJ2CActivationSpecs(ra, '[-messageListener Type javax.jms.MessageListener]')
```

- Using Jython list:

```
AdminTask.listJ2CActivation Specs(ra, ['-messageListener Type', 'javax.jms.MessageListener'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listJ2CActivationSpecs {-interactive}
```

- Using Jython:

```
AdminTask.listJ2CActivationSpecs('-interactive')
```

listJ2CAdminObjects

Use the listJ2CAdminObjects command to list administrative objects that contain the administrative object interface that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

-adminObjectInterface
Specifies the administrative object interface for which you want to list. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listJ2CAdminObjects $ra {-adminObjectInterface
 fvt.adaptor.message.FVTMessageProvider}
```

- Using Jython string:

```
AdminTask.listJ2CAdminObjects(ra, '[-adminObjectInterface
 fvt.adaptor.message.FVTMessageProvider]')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listJ2CAdminObjects {-interactive}
```

- Using Jython:

```
AdminTask.listJ2CAdminObjects('-interactive')
```

listJ2CConnectionFactories

Use the listJ2CConnectionFactories command to list the Java 2 connector connection factories under the resource adapter and connection factory interface that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

-connectionFactoryInterface
Indicates the name of the connection factory that you want to list. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listJ2CConnectionFactories $ra {-connectionFactoryInterface javax.sql.DataSource}
```

- Using Jython string:

```
AdminTask.listJ2CConnectionFactories(ra, '[-connectionFactoryInterface javax.sql.DataSource]')
```

- Using Jython list:

```
AdminTask.listJ2CConnectionFactories(ra, ['-connectionFactoryInterface',
 'javax.sql.DataSource'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listJ2CConnectionFactories {-interactive}
```

- Using Jython:

```
AdminTask.listJ2CConnectionFactories('-interactive')
```

listMessageListenerTypes

Use the listMessageListenerTypes command to list the message listener types that are defined under the resource adapter that you specify.

Target object

J2C Resource Adapter object ID

Parameters and return values

- Parameters: None
- Returns: A list of message listener types.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listMessageListenerTypes $ra
```

- Using Jython string:

```
AdminTask.listMessageListenerTypes(ra)
```

- Using Jython list:

```
AdminTask.listMessageListenerTypes(ra)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listMessageListenerTypes {-interactive}
```

- Using Jython:

```
AdminTask.listMessageListenerTypes('-interactive')
```

Related tasks

“Configuring new J2C administrative objects using scripting” on page 961
You can use scripting and the wsadmin tool to configure new J2C administrative objects.

“Configuring new J2C activation specifications using scripting” on page 959
You can configure new J2C activation specifications using scripting and the wsadmin tool.

“Configuring new J2C resource adapters using scripting” on page 955
Use the wsadmin tool to configure resource adapters with Resource Adapter Archive (RAR) files. A RAR file provides the classes and other code to support a resource adapter for access to a specific enterprise information system (EIS), such as the Customer Information Control System (CICS). Configure resource adapters for an EIS only after you install the appropriate RAR file.

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 13. Configuring mail, URLs, and resource environment entries with scripting

Use scripting to configure mail, URLs, and resource environment entries.

About this task

This topic contains the following tasks:

Configuring new mail providers using scripting

You can use scripting and the wsadmin tool to configure new mail providers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 77 article for more information.

About this task

Perform the following steps to configure a new mail provider:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
     ```
   - Using Jython:
     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required MailProvider
     ```
   - Using Jython:
     ```
     print AdminConfig.required('MailProvider')
     ```

   Example output:

```
Attribute       Type
name       String
```

3. Set up required attributes:
   - Using Jacl:
     ```
     set name [list name MP1]
     set mpAttrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'MP1']
     mpAttrs = [name]
     ```
4. Create the mail provider:
   - Using Jacl:
     ```
     set newmp [$AdminConfig create MailProvider $node $mpAttrs]
     ```
   - Using Jython:
     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new mail sessions using scripting

You can use scripting and the wsadmin tool to configure new mail sessions.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new mail session:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```
   - Using Jython:
     ```
     newmp = AdminConfig.getid('/Cell:mycell/Node:mynode/MailProvider:MP1/')
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required MailSession
     ```
   - Using Jython:
     ```
     print AdminConfig.required('MailSession')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName       String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MS1]
     set jndi [list jndiName mail/MS1]
     set msAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name MS1} {jndiName mail/MS1}
     ```

   - Using Jython:

     ```
     name = ['name', 'MS1']
     jndi = ['jndiName', 'mail/MS1']
     msAttrs = [name, jndi]
     print msAttrs
     ```

     Example output:

     ```
     [[name, MS1], [jndiName, mail/MS1]]
     ```

4. Create the mail session:
   - Using Jacl:

     ```
     $AdminConfig create MailSession $newmp $msAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('MailSession', newmp, msAttrs)
     ```

   Example output:

   ```
   MS1(cells/mycell/nodes/mynode|resources.xml#MailSession_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new protocols using scripting

You can configure new protocols with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new protocol:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required ProtocolProvider
     ```

   - Using Jython:

     ```
     print AdminConfig.required('ProtocolProvider')
     ```

Example output:

```
Attribute       Type
protocol      String
classname       String
```

3. Set up required attributes:
   - Using Jacl:

     ```
     set protocol [list protocol "Put the protocol here"]
     set classname [list classname "Put the class name here"]
     set ppAttrs [list $protocol $classname]
     ```

     Example output:

     ```
     {protocol protocol1} {classname classname1}
     ```

   - Using Jython:

     ```
     protocol = ['protocol', "Put the protocol here"]
     classname = ['classname', "Put the class name here"]
     ppAttrs = [protocol, classname]
     print ppAttrs
     ```

     Example output:

     ```
     [[protocol, protocol1], [classname, classname1]]
     ```

4. Create the protocol provider:
   - Using Jacl:

     ```
     $AdminConfig create ProtocolProvider $newmp $ppAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('ProtocolProvider', newmp, ppAttrs)
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#ProtocolProvider_4)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new custom properties using scripting

You can use scripting and the wsadmin tool to configure new custom properties.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new custom property:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```

2. Get the J2EE resource property set:

- Using Jacl:

  ```
  set propSet [$AdminConfig showAttribute $newmp propertySet]
  ```

- Using Jython:

  ```
  propSet = AdminConfig.showAttribute(newmp, 'propertySet')
  print propSet
  ```

  Example output:

  ```
  (cells/mycell/nodes/mynode|resources.xml#PropertySet_2)
  ```

3. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute       Type
   name      String
   ```

4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name CP1]
     set cpAttrs [list $name]
     ```

     Example output:

     ```
     {name CP1}
     ```

   - Using Jython:

     ```
     name = ['name', 'CP1']
     cpAttrs = [name]
     print cpAttrs
     ```

     Example output:

     ```
     [[name, CP1]]
     ```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $cpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, cpAttrs)
     ```

   Example output:

   ```
   CP1(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_2)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new resource environment providers using scripting

You can use the wsadmin tool and scripting to configure new resources environment providers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new resource environment provider:

1. Identify the parent ID and assign it to the node variable.
   - Using Jacl:
     ```
     set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
     ```
   - Using Jython:
     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:
   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```
2. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required ResourceEnvironmentProvider
     ```
   - Using Jython:
     ```
     print AdminConfig.required('ResourceEnvironmentProvider')
     ```

   Example output:
   ```
   Attribute     Type
   name     String
   ```
3. Set up the required attributes and assign it to the repAttrs variable:
   - Using Jacl:
     ```
     set n1 [list name REP1]
     set repAttrs [list $name]
     ```
   - Using Jython:
     ```
     n1 = ['name', 'REP1']
     repAttrs = [n1]
     ```
4. Create a new resource environment provider:
   - Using Jacl:
     ```
     set newrep [$AdminConfig create ResourceEnvironmentProvider $node $repAttrs]
     ```
   - Using Jython:
     ```
     newrep = AdminConfig.create('ResourceEnvironmentProvider', node, repAttrs)
     print newrep
     ```

   Example output:
   ```
   REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring custom properties for resource environment providers using scripting

You can use scripting to configure custom properties for a resource environment provider.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new custom property for a resource environment provider:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:

     ```
     set newrep [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
     ```

   - Using Jython:

     ```
     newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
     print newrep
     ```

   Example output:

   ```
   REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
   ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute     Type
   name     String
   ```

3. Set up the required attributes and assign it to the repAttrs variable:
   - Using Jacl:

     ```
     set name [list name RP]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP']
     rpAttrs = [name]
     ```

4. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newrep propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newrep, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_1)
   ```

   If the command returns `None` as the value for the propSet variable, create a new property set. The command returns `None` if the property set does not exist in your environment. Use the following examples to create a new property set:

   Using Jacl:

   ```
   set newPropSet  [$AdminConfig  create  $newrep {}]
   ```

   Using Jython:

   ```
   newPropSet = AdminConfig.create('J2EEResourcePropertySet',newrep,[])
   ```

   After setting the newPropSet variable, retry the command to get the J2EE resource property set before going to the next step.

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

```
RP(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
```

6. Save the configuration changes.

   Using Jacl:

   ```
   $AdminConfig save
   ```

   Using Jython:

   ```
   AdminConfig.save()
   ```

7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new referenceables using scripting

You can use scripting and the wsadmin tool to configure new referenceables.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new referenceable:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:

     ```
     set newrep [$AdminConfig  getid  /Cell:mycell/Node:mynode/
     ResourceEnvironmentProvider:REP1/]
     ```

   - Using Jython:

     ```
     newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/
     ResourceEnvironmentProvider:REP1/')
     print newrep
     ```

   Example output:

   ```
   REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
   ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required Referenceable
     ```

   - Using Jython:

     ```
     print AdminConfig.required('Referenceable')
     ```

   Example output:

   ```
   Attribute     Type
   factoryClassname   String
   classname   String
   ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set fcn [list factoryClassname REP1]
     set cn [list classname NM1]
     set refAttrs [list $fcn $cn]
     ```

   - Using Jython:

     ```
     fcn = ['factoryClassname', 'REP1']
     cn = ['classname', 'NM1']
     refAttrs = [fcn, cn]
     print refAttrs
     ```

   Example output:

   ```
   {factoryClassname {REP1}} {classname {NM1}}
   ```

4. Create a new referenceable:
   - Using Jacl:
     ```
     set newref [$AdminConfig create Referenceable  $newrep  $refAttrs]
     ```
   - Using Jython:
     ```
     newref = AdminConfig.create('Referenceable',  newrep,  refAttrs)
     print newref
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#Referenceable_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new resource environment entries using scripting

You can use scripting and the wsadmin tool to configure a new resource environment entry.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new resource environment entry:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:
     ```
     set newrep [$AdminConfig  getid  /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
     ```
   - Using Jython:
     ```
     newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
     print newrep
     ```

   Example output:

   ```
   REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
   ```
2. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required ResourceEnvEntry
     ```
   - Using Jython:
     ```
     print AdminConfig.required('ResourceEnvEntry')
     ```

   Example output:

   ```
   Attribute     Type
   name      String
   jndiName   String
   referenceable   Referenceable@
   ```
3. Set up the required attributes:
   - Using Jacl:
     ```
     set name [list name REE1]
     set jndiName [list jndiName myjndi]
     set newref [$AdminConfig getid /Cell:mycell/Node:mynode/Referenceable:/]
     set ref [list referenceable $newref]
     set reeAttrs [list $name $jndiName $ref]
     ```
   - Using Jython:

```
name = ['name', 'REE1']
jndiName = ['jndiName', 'myjndi']
newref = AdminConfig.getid('/Cell:mycell/Node:mynode/Referenceable:/')
ref = ['referenceable', newref]
reeAttrs = [name, jndiName, ref]
```

4. Create the resource environment entry:
   - Using Jacl:

     ```
     $AdminConfig create ResourceEnvEntry $newrep $reeAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('ResourceEnvEntry', newrep, reeAttrs)
     ```

   Example output:

   ```
   REE1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvEntry_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring custom properties for resource environment entries using scripting

You can use scripting to configure a new custom property for a resource environment entry.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new custom property for a resource environment entry:

1. Identify the parent ID and assign it to the newree variable.
   - Using Jacl:

     ```
     set newree [$AdminConfig  getid  /Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/]
     ```

   - Using Jython:

     ```
     newree = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/')
     print newree
     ```

   Example output:

   ```
   REE1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvEntry_1)
   ```

2. Create the J2EE custom property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newree propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newree, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_5)
   ```

3. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

Example output:

```
Attribute     Type
name      String
```

4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name RP1]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP1']
     rpAttrs = [name]
     ```

5. Create the J2EE custom property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RPI(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring new URL providers using scripting

You can use scripting and the wsadmin tool to configure new URL providers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new URL provider:

1. Identify the parent ID and assign it to the node variable.
   - Using Jacl:

     ```
     set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
     ```

   - Using Jython:

     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required URLProvider
     ```

   - Using Jython:

     ```
     print AdminConfig.required('URLProvider')
     ```

   Example output:

```
Attribute     Type
streamHandlerClassName   String
protocol       String
name         String
```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name URLP1]
     set shcn [list streamHandlerClassName "Put the stream handler classname here"]
     set protocol [list protocol "Put the protocol here"]
     set urlpAttrs [list $name $shcn $protocol]
     ```

     Example output:

     ```
     {name URLP1} {streamHandlerClassName {Put the stream handler classname here}}
     {protocol {Put the protocol here}}
     ```

   - Using Jython:

     ```
     name = ['name', 'URLP1']
     shcn = ['streamHandlerClassName', "Put the stream handler classname here"]
     protocol = ['protocol', "Put the protocol here"]
     urlpAttrs = [name, shcn, protocol]
     print urlpAttrs
     ```

     Example output:

     ```
     [[name, URLP1], [streamHandlerClassName, "Put the stream handler classname here"],
     [protocol, "Put the protocol here"]]
     ```

4. Create a URL provider:
   - Using Jacl:

     ```
     $AdminConfig create URLProvider $node $urlpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('URLProvider', node, urlpAttrs)
     ```

   Example output:

   ```
   URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring custom properties for URL providers using scripting

You can use scripting to configure custom properties for URL providers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure custom properties for URL providers:

1. Identify the parent ID and assign it to the newurlp variable.
   - Using Jacl:

     ```
     set newurlp [$AdminConfig  getid  /Cell:mycell/Node:mynode/URLProvider:URLP1/]
     ```

   - Using Jython:

     ```
     newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
     print newurlp
     ```

   Example output:

```
URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
```

2. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newurlp propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newurlp, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_7)
   ```

3. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute     Type
   name         String
   ```

4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name RP2]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP2']
     rpAttrs = [name]
     ```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP2(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Related tasks

“Configuring custom properties for URLs using scripting” on page 1010
Use the wsadmin tool and scripting to set custom properties for URLs.

Configuring new URLs using scripting

You can use scripting and the wsadmin tool to configure new URLs.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following example to configure a new URL:

1. Identify the parent ID and assign it to the newurlp variable.
   - Using Jacl:

     ```
     set newurlp [$AdminConfig  getid  /Cell:mycell/Node:mynode/URLProvider:URLP1/]
     ```

   - Using Jython:

     ```
     newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
     print newurlp
     ```

     Example output:

     ```
     URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
     ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required URL
     ```

   - Using Jython:

     ```
     print AdminConfig.required('URL')
     ```

     Example output:

     ```
     Attribute     Type
     name          String
     spec          String
     ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name URL1]
     set spec [list spec "Put the spec here"]
     set urlAttrs [list $name $spec]
     ```

     Example output:

     ```
     {name URL1} {spec {Put the spec here}}
     ```

   - Using Jython:

     ```
     name = ['name', 'URL1']
     spec = ['spec', "Put the spec here"]
     urlAttrs = [name, spec]
     ```

     Example output:

     ```
     [[name, URL1], [spec, "Put the spec here"]]
     ```

4. Create a URL:
   - Using Jacl:

     ```
     $AdminConfig create URL $newurlp $urlAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('URL', newurlp, urlAttrs)
     ```

     Example output:

     ```
     URL1(cells/mycell/nodes/mynode|resources.xml#URL_1)
     ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Configuring custom properties for URLs using scripting

Use the wsadmin tool and scripting to set custom properties for URLs.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to configure a new custom property for a URL:

1. Identify the parent ID and assign it to the newurl variable.
   - Using Jacl:

     ```
     set newurl [$AdminConfig getid /Cell:mycell/Node:mynode/URLProvider:URLP1/URL:URL1/]
     ```
   - Using Jython:

     ```
     newurl = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/URL:URL1/')
     print newurl
     ```

   Example output:

   ```
   URL1(cells/mycell/nodes/mynode|resources.xml#URL_1)
   ```
2. Create a J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newurl propertySet]
     ```
   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newurl, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_7)
   ```
3. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```
   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute     Type
   name         String
   ```
4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name RP3]
     set rpAttrs [list $name]
     ```
   - Using Jython:

     ```
     name = ['name', 'RP3']
     rpAttrs = [name]
     ```
5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```
   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP3(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_7)
   ```
6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.

7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Provider command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure mail settings with the wsadmin tool. The commands and parameters in the provider group can be used to create, delete, and manage providers.

The Provider command group for the AdminTask object includes the following commands:

- “deleteProvider”
- “getProviderInfo”
- “getProviderInstance” on page 1013
- “getProviders” on page 1013

deleteProvider

The **deleteProvider** command deletes a provider from the model given the providerName.

Target object

None

Parameters and return values

- Parameters: None
- Returns: true if the provider was deleted, false if not.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteProvider {-interactive}
```

- Using Jython string:

```
AdminTask.deleteProvider ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteProvider (['-interactive'])
```

getProviderInfo

The **getProviderInfo** command returns a ProviderInfo object defined in the model given the providerName.

Target object

None

Parameters and return values

- Parameters: None
- Returns: A ProviderInfo object defined in the model given the providerName.

Examples

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getProviderInfo {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getProviderInfo ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getProviderInfo (['-interactive'])
  ```

getProviderInstance

The **getProviderInstance** command returns a java.security.Provider for the providerName specified.

Target object

None

Parameters and return values
- Parameters: None
- Returns: A java.security.Provider for the providerName specified.

Examples

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask getProviderInsta nce {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getProviderIns tance ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getProviderInst ance (['-interactive'])
  ```

getProviders

The **getProviders** command returns a List of ProviderInfo objects from the model.

Target object

None

Parameters and return values
- Parameters: None
- Returns: A list of ProviderInfo objects from the model.

Examples

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask getProviders {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getProviders ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getProviders (['-interactive'])
  ```

Related tasks

"Using the AdminTask object for scripted administration" on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

"Commands for the AdminTask object" on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 14. Configuring Web services applications using scripting

You can use the wsadmin scripting tool to complete the several tasks for a Web services application.

Before you begin

Develop a Web services application.

About this task

The WebSphere Application Server wsadmin tool provides the ability to run scripts. You can use the wsadmin tool to manage a WebSphere Application Server installation, as well as configuration, application deployment, and server run-time operations. The WebSphere Application Server only supports the Jacl and Jython scripting languages.

To use the wsadmin tool to configure a Web services application, publish a Web Services Description Language (WSDL) file, or to configure policy sets:

1. Launch a scripting command.
2. Follow the steps in one of the following topics, depending on what task you want to complete:
   - Publish WSDL files.
   - Configure Web service client deployed WSDL file names.
   - Configure Web service client bindings.
   - Configure Web service client preferred port mappings .
   - Configure Web service client port information .
   - Configure the scope of a Web service port .

Results

You have configured Web services applications with the wsadmin tool.

Enabling WSDM with scripting

Use the wsadmin tool with the AdminConfig object to enable Web Services Distributed Management (WSDM) in your environment. WSDM is an OASIS approved standard that supports managing resources through a standardized Web service interface.

About this task

The WSDM application is installed as a system application and is disabled by default. In order to use the WSDM functionality, use the script in this topic to enable WSDM.

In a multinode environment, the management code runs across a distributed network of Java virtual machines with a central access point as the deployment manager process for the entire network or cell. Enable WSDM on the deployment manager to manage Java virtual machines within a cell. The WSDM application acts as an administrative client to the managed application server. You can manage the WSDM application from the application server on which it is deployed only.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the configuration ID of the WSDM application.

   Use the getid option for the AdminConfig object to retrieve the configuration ID and set the value to the `deployment` variable, as the following example demonstrates:

```
deployment = AdminConfig.getid('/Deployment:WebSphereWSDM/')
```

3. Determine the deployed object of the WSDM configuration ID.

   Use the showAttribute option for the AdminConfig object to retrieve the deployedObject attribute and set the value to the `deployedObject` variable, as the following example demonstrates:

```
deployedObject = AdminConfig.showAttribute(deployment, 'deployedObject')
```

4. Determine the target mappings for the WSDM deployed object.

   Use the showAttribute option for the AdminConfig object to retrieve the targetMappings attribute and set the value to the targetMappings variable, as the following example demonstrates:

```
targetMappings = AdminConfig.showAttribute(deployedObject, "targetMappings")
```

5. Enable WSDM.

   Assign each mapping to the `target` variable, then set the enable attribute to `true` in the target mapping, as the following example demonstrates:

```
mappings = targetMappings[1:len(targetMappings)-1].split(" ")
AdminConfig.modify(target, '[[enable true]]')
```

6. Save the configuration changes.

Example

The following samples provide Jython and Jacl scripts that enable WSDM in your environment:

- Using Jython:

```
deployment = AdminConfig.getid('/Deployment:WebSphereWSDM/')
deployedObject = AdminConfig.showAttribute(deployment, 'deployedObject')
targetMappings = AdminConfig.showAttribute(deployedObject, "targetMappings")
mappings = targetMappings[1:len(targetMappings)-1].split(" ")
for target in mappings:
AdminConfig.modify(target, '[[enable true]]')
AdminConfig.save()
```

- Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:WebSphereWSDM]
set deployedObject [$AdminConfig showAttribute $deployment deployedObject]
set targetMappings [lindex [$AdminConfig showAttribute $deployedObject targetMappings] 0]
$AdminConfig modify $targetMappings {{enable true}}
$AdminConfig save
```

Querying Web services with the wsadmin tool

You can use the Jython or Jacl scripting language to query for Web services properties with the wsadmin tool. Use the commands in the WebServicesAdmin group to list all Web services and attributes, find attributes of a specific Web service, determine the Web service endpoint, and determine the operation name of a Web service.

Before you begin

The wsadmin administrative scripting program supports two scripting languages, Jacl and Jython. The Jacl syntax is deprecated. This topic includes examples written only in the Jython scripting language.

Before you can complete the procedure for the commands in the WebServicesAdmin group, you must launch the wsadmin tool.

About this task

Use the following commands to query for Web services and Web service attributes. If you receive a NoItemFoundException error, the specified application, module, service, or endpoint cannot be found. Verify that all input parameters are correct.

You can optionally specify the client parameter for any command in the WebServicesAdmin command group. The client parameter indicates whether to return service providers or service clients. Specify `false` to request service providers and `true` to request service clients.

- Query the configuration for all installed Web services for all enterprise applications.

  Enter the following command. You do not need to specify the application parameter for this command.

```
AdminTask.listWebServices()
```

  This command returns all installed Web services. The command also returns the application name, module name, service name, and service type for each Web service.

  Sample output:

```
'[ [service {http://www.ibm.com}service1] [client false] [application
application1] [module webapp1.war] [type JAX-WS] ]'
```

- Query the configuration for all installed Web services for a specific enterprise application.

  Enter the following command, and specify the name of the application you want to query:

```
AdminTask.listWebServices('[–application application_name -client false]')
```

  This command returns all installed Web services for the *application_name* that you specify. The command also returns the application name, module name, service name, and service type for each Web service.

  Sample output:

```
'[ [service {http://www.ibm.com}service1] [client false] [application
application1] [module webapp1.war] [type JAX-WS] ]'
```

- Query the configuration for the Web service name and type for a enterprise application.

  Enter the following command, and specify the application name, module name, and Web service name. The **client** parameter is optional.

```
AdminTask.getWebService('[–application application_name
-module module_name -service webservice_name -client false]')
```

  The command returns the Web service name and Web service type.

  Sample output:

```
'[ [service {http://www.ibm.com}service1] [client false] [type JAX-WS] ]'
```

- Query the configuration for the Web service endpoints for a enterprise application.

  The logical endpoint name is the port name in the Web Services Description Language (WSDL) document. Enter the following command, and specify the application name, module name, and Web service name. The **client** parameter is optional.

```
AdminTask.listWebServiceEndpoints('[–application application_name
 -module module_name -service webservice_name -client false]')
```

  This command returns the port on which the Web service is installed.

  Sample output:

```
'[logicalEndpoint QuotePort01]'
```

- Query the configuration for the Web service operation names for a enterprise application.

  Enter the following command, and specify the application name, module name, Web service name, and endpoint name. The logical endpoint name is the port name in the Web Services Description Language (WSDL) document. The **client** parameter is optional.

```
AdminTask.listWebServiceOperations('[–application application_name -module module_name
-service webservice_name -logicalEndpoint endpoint_name -client false]')
```

  This command returns all Web service operations.

  Sample output:

```
'[operation ivt_app_op1] [operation ivt_app_op2]'
```

- Query the configuration for the service providers, endpoints, and operations from each deployed asset.

  The listServices command provides generic query functions. Use the following command to display information about service providers, endpoints, and operations for enterprise applications and Web Services Notification (WSN) clients. Each parameter is optional. If you do not specify the queryProps parameter, the command returns each service provider in your configuration. If you do not specify the

expandResources parameter, the command does not return the logical endpoints or operations for each service. The following command example returns each service provider and the corresponding endpoints for the `myApplication` application:

```
AdminTask.listServices('-queryProps "[[CompositionUnit=myApplication][client=false]" -expandResources endpoint']
```

The following command example returns each JAX-WS service provider and the corresponding endpoints and operations within a cell:

```
AdminTask.listServices('[-queryProps "[[serviceType JAX-WS][client false]]"]')
```

This command returns the service providers that match the search query.

Sample output:

```
'[ [service {http://com/ibm/was/wssample/sei/echo/}EchoService] [assetType [J2EE Application]] [client false] [application WSSampleServicesSei]
 [module SampleServicesSei.war] [serviceType JAX-WS] ]

[ [service {http://com/ibm/was/wssample/sei/echo/}EchoService12] [assetType [J2EE Application]] [client false] [application WSSampleServicesSei]
 [module SampleServicesSei.war] [serviceType JAX-WS] ]

[service {http://com/ibm/was/wssample/sei/ping/}PingService] [assetType [J2EE Application]] [client false] [application WSSampleServicesSei]
 [module SampleServicesSei.war] [serviceType JAX-WS] ]

[ [service {http://com/ibm/was/wssample/sei/ping/}PingService12] [assetType [J2EE Application]] [client false] [application WSSampleServicesSei]
 [module SampleServicesSei.war] [serviceType JAX-WS]]'
```

The following command example returns each WSN service client and the corresponding endpoints and operations within a cell:

```
AdminTask.listServices('[-queryProps "[[serviceType [JAX-WS (WSN)]][clie
nt true]]" -expandResource logicalEndpoint]')
```

This command returns the service providers that match the search query.

Sample output:

```
'[ [service {http://www.ibm.com/websphere/wsn/out/remote-publisher}OutboundRemotePublisherService] [assetType [WSN Service]] [client true]
 [bus bus1] [WSNService wsn1] [serviceType [JAX-WS (WSN)]] ]

[ [assetType [WSN Service]] [service {http://www.ibm.com/websphere/wsn/out/remote-publisher}OutboundRemotePublisherService]
 [bus bus1] [client true] [WSNService wsn1] [serviceType [JAX-WS (WSN)]] [logicalEndpoint OutboundRemotePublisherPort] ]

[ [service {http://www.ibm.com/websphere/wsn/out/notification}OutboundNotificationService] [assetType [WSN Service]] [client true]
 [bus bus1] [WSNService wsn1] [serviceType [JAX-WS (WSN)]] ]

[ [assetType [WSN Service]] [service {http://www.ibm.com/websphere/wsn/out/notification}OutboundNotificationService]
 [bus bus1] [client true] [WSNService wsn1] [serviceType [JAX-WS (WSN)]] [logicalEndpoint OutboundNotificationPort] ]'
```

Related tasks

“Starting the wsadmin scripting client” on page 77
You can use the wsadmin tool to configure and administer application servers, application deployment, and server run-time operations.

“Configuring application and system policy sets for Web services using scripting” on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

Chapter 1, “Using scripting (wsadmin),” on page 1
The WebSphere administrative (wsadmin) scripting program is a powerful, non-graphical command interpreter environment enabling you to run administrative operations in a scripting language.

WebServicesAdmin command group for the AdminTask object

Use this topic as a reference for the commands for the WebServicesAdmin group of the AdminTask object. Use these commands with your administrative scripts to list available Web services, list web services attributes, determine the endpoint configuration for a Web service, and determine a specific operation name.

Use the commands in the WebServicesAdmin group to query information for installed web services. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WebServicesAdmin group of the AdminTask object.

getWebService

The getWebService command retrieves the attributes for a Web service. This command applies to enterprise applications only.

Target object

None.

Required parameters

-application
Specifies the name of the deployed enterprise application. (String, required)

-module
Specifies the name of the module. (String, required)

-service
Specifies the name of the Web service. (String, required)

Optional parameters

-client
Specifies whether the Web service is a provider or a client. The default value is `false` (service provider). When set to `true`, the command only returns Web service clients. (Boolean, optional)

Return value

Returns a list of attributes, including service name, whether the service is a provider or client, and service type. The service type attribute is only applicable for service providers.

Batch mode example usage

Using Jython string:

```
AdminTask.getWebService('[–application application_name -module module_name -service webservice_name -client false]')
```

Sample output:

```
'[[service {http://www.ibm.com}service1][type JAX-WS][client false]]'
```

Interactive mode example usage

Using Jython:

```
AdminTask.getWebService ('-interactive')
```

listServices

The listServices command queries the configuration for services, endpoints, and operations. This command provides more generic query functions than the rest of commands in this command group. It is applicable to Java™ applications as well as other assets such as Web Services Notification (WSN) clients.

Target object

None.

Optional parameters

-queryProps
Specifies properties used to locate the service provider or client of interest. For example, if you specify `[[type=JAX-WS][client=true]]`, the command returns each JAX-WS reference in the enterprise applications. (Properties, optional)

The -queryProps parameter accepts the several properties. You can use one or multiple properties to specify your query criteria. Do not mix the query properties between different asset types. For example, if you specify `application` and `bus`, the command reports an error.

- Specify the following properties with the -queryProps parameter to query enterprise applications:

Property and value	Description
`assetType=J2EE Application`	Queries each Java EE application.
`application=`*`application_name`*	Queries a specific Java EE application.
`module=`*`module_name`*	Queries a specific Java EE application module. You must specify the `application` and `module` properties to query for application modules.

- Specify the following properties with the -queryProps parameter to query for WSN clients:

Property and value	Description
`assetType=WSN Service`	Queries each WSN service client.
`bus=`*`bus_name`*	Queries a specific bus.
`WSNService=`*`WSN_service_name`*	Queries a specific WSN service. You must specify the `bus` and `WSNService` properties to query for a specific WSN service.

- Specify the following properties with the -queryProps parameter to query all assets:

Property and value	Description
`serviceType=`*`service_type`*	Queries by service type. Specify `JAX-WS` to query for Java™ API for XML-Based Web Services assets. Specify `JAX-WS (WSN)` to query for Web Services Notification assets.
`client=`*`Boolean`*	Queries for clients or providers. Specify `true` to query for clients. Specify `false` to query for providers.
`service=`*`service_name`*	Queries for the logical endpoints and operations for a specific service.

-expandResource
Specifies whether to return service names only or services and detailed resource information. Specify `logicalEndpoint` or `operation`. If you specify the `logicalEndpoint` value, the command returns the matching services and each endpoint in the services. If you specify the `operation` value, the command returns the matching services and the corresponding endpoints and operations. (String, optional)

Return value

The command returns a list of properties for each service, as well as detailed endpoint and operation information if you query for endpoints and operations.

Batch mode example usage

The following examples query each service provider in the myApplication application. The examples return the logical endpoints for each service because the -expandResources parameter is set as logicalEndpoint, as the following example output demonstrates:

```
[ [service {http://com/ibm/was/wssample/sei/echo/}EchoService] [assetType [J2EE Application]]
 [client false] [application MyWSApplication] [module ServicesModule.war] [serviceType JAX-WS] ]
[ [assetType [J2EE Application]] [service {http://com/ibm/was/wssample/sei/echo/}EchoService]
 [client false] [application MyWSApplication] [module ServicesModule.war] [serviceType JAX-WS] [logicalEndpoint EchoServicePort] ]

[ [service {http://com/ibm/was/wssample/sei/ping/}PingService] [assetType [J2EE Application]]
 [client false] [application MyWSApplication] [module ServicesModule.war] [serviceType JAX-WS] ]
[ [assetType [J2EE Application]] [service {http://com/ibm/was/wssample/sei/ping/}PingService]
 [client false] [application MyWSApplication] [module ServicesModule.war] [serviceType JAX-WS]
 [logicalEndpoint PingServicePort] ]
```

Using Jython string:

```
AdminTask.listServices('[-queryProps [[application MyWSApplication][client false]] -expandResource logicalEndpoint]')
```

Using Jython list:

```
AdminTask.listServices(['–queryProps', '[[application myApplication][client false]]', '-expandResource', 'logicalEndpoint'])
```

The following examples query each service client under the myBus bus. The examples do not return logical endpoints or operations for each service client because it does not specify the -expandResource parameter.

Using Jython string:

```
AdminTask.listServices('[–queryProps [[bus myBus][client true]] ]')
```

Using Jython list:

```
AdminTask.listServices(['–queryProps', '[[bus myBus][client true]]'])
```

Interactive mode example usage

Using Jython:

```
AdminTask.listServices('-interactive')
```

listWebServices

The listWebServices command retrieves a list of available Web services for one or all applications. If an application name is not supplied, the command lists all of the Web services. This command applies to enterprise applications only.

Target object

None.

Required parameters

None.

Optional parameters

-application
Specifies the name of the deployed enterprise application. If you do not specify this parameter, the command returns all Web services in the cell. (String, optional)

-client
Specifies whether the Web service is a provider or a client. The default value is false (service provider). When set to true, the command only returns Web service clients. (Boolean, optional)

Return value

All Web services for the application specified. For each Web service, the command returns the following attributes and corresponding values: application name, module name, service name, whether the Web service is a service provider or client, and service type. The service type is only specified if the Web service is a service provider.

Batch mode example usage

Using Jython string:

```
AdminTask.listWebServices('[–application application1 -client false]')
```

Using Jython list:

```
AdminTask.listWebServices(['–application', 'application1', '-client', 'false'])
```

Sample output:

```
'[[service {http://www.ibm.com}service1][application application1][module webapp1.war][type JAX-WS][client false]]'
```

Interactive mode example usage

Using Jython:

```
AdminTask.listWebServices('-interactive')
```

listWebServicesEndpoints

The listWebServicesEndpoints command returns a list of logical endpoints for a Web service. The logical endpoint name is the port name in the Web Services Description Language (WSDL) document. This command applies to enterprise applications only.

Target object

None.

Required parameters

-application
Specifies the name of the deployed enterprise application. (String, required)

-module
Specifies the name of the module. (String, required)

-service
Specifies the name of the Web service. (String, required)

Optional parameters

-client
Specifies whether the Web service is a provider or a client. The default value is `false` (service provider). When set to `true`, the command only returns Web service clients. (Boolean, optional)

Return value

Returns the logical endpoint name for the Web service specified.

Batch mode example usage

Using Jython string:

```
AdminTask.listWebServiceEndpoints('[–application application_name -module module_name -service webservice_name -client false]')
```

Using Jython list:

```
AdminTask.listWebServiceEndpoints(['–application', 'application_name', '-module', 'module_name', '-service', 'webservice_name', '-client', 'false'])
```

Sample output:

```
'[[logicalEndpoint QuotePort01]]'
```

Interactive mode example usage

Using Jython:

```
AdminTask.listWebServicesEndpoints('-interactive')
```

listWebServicesOperations

The listWebServicesOperations command returns a list of Web service operations. This command applies to enterprise applications only.

Target object

None.

Required parameters

-application
Specifies the name of the deployed enterprise application. (String, required)

-module
Specifies the name of the module. (String, required)

-service
Specifies the name of the Web service. (String, required)

-logicalEndpoint
The port name in the Web Services Description Language (WSDL) document. (String, required)

Optional parameters

-client
Whether the Web service is a provider or a client. The default value is `false` (service provider). When set to `true`, the command only returns Web service clients. (Boolean, optional)

Return value

Returns the operation name for the Web service specified.

Batch mode example usage

Using Jython string:

```
AdminTask.listWebServiceOperations('[–application application_name -module module_name -service webservice_name -client false
 -logicalEndpoint endpoint_name]')
```

Using Jython list:

```
AdminTask.listWebServiceOperations(['–application', 'application_name', '-module', 'module_name', '-service', 'webservice_name', '-client',
 'false', '-logicalEndpoint', 'endpoint_name'])
```

Sample output:

```
'[[operation ivt_app_op1][operation ivt_app_op2]]'
```

Interactive mode example usage

Using Jython:

```
AdminTask.listWebServicesOperations('-interactive')
```

Configuring a Web service client deployed WSDL file name with the wsadmin tool

When a Web service application is deployed into WebSphere Application Server, an instance is created for each application or module. The instance contains deployment information for the Web module or Enterprise JavaBeans (EJB) module, including the client bindings.

Before you begin

You should have the Web service application ready for deployment or already deployed the Web service into WebSphere Application Server before starting this task.

To complete this task, you need to know the topology of the URL endpoint address of the Web services servers and which Web service the client depends upon. You can view the deployment descriptors in the administrative console to find topology information. See the article View Web services server deployment descriptors for more information.

For more information about the wsadmin tool options, review Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands

About this task

The client bindings define the Web Services Description Language (WSDL) file name and preferred ports. The relative path of a Web service in a module is specified within a compatible WSDL file that contains the actual URL to be used for requests. The address is only needed if the original WSDL file did not contain a URL, or when a different address is needed. For a service endpoint with multiple ports, you need to define an alternative WSDL file name.

The following steps describe how to edit bindings for a Web service after these bindings are deployed on a server. When one Web service communicates with another Web service, you must configure the client bindings to access the downstream Web service.

You can use the **WebServicesClientBindDeployedWsdl** command-line option in this task to change the endpoint. One of the benefits to using the command-line option is that you can avoid uninstalling, modifying enterprise archive (EAR) files and reinstalling applications to make binding configuration changes. Another benefit is the ability to customize the Web service bindings applications for different environments during installation, and to avoid the need to create different application EAR files for each version.

Several versions of WSDL files, each with different service endpoints, can be provided during the development and assembly of a Web service module that is acting as a client to a Web service. During or after the installation, when you are configuring the installed application, the WebServicesClientBindDeployedWsdl option can be used to specify which of the WSDL files to use.

Because the WSDL file defines all the service endpoints or implementations for all of the port types and ports that the client can use, the deployed WSDL file can group a set of choices into one WSDL. You can override the endpoint by port.

You can use Jacl or Jython scripts, but this task assumes that you are using Jacl. For more information about using scripting see Deploying and managing using scripting.

To configure client bindings with the wsadmin tool, proceed with the following steps:

1. Launch a scripting command.
2. At the **wsadmin** command prompt, enter the command syntax. You can use install, installInteractive, edit or editInteractive options. The following example presents the syntax:

```
$AdminApp edit <app_name> {
-WebServicesClientBindDeployedWsdl {{<module_name> <EJB_name> <web_service>
<deployed_WSDL_filename>}...}
```

The example shows multiple nodules and URL endpoints because you can edit multiple URL fragments. where:

- *app_name* is the application name, for example `WebServicesSample.ear`
- *module_name* is the EJB or Web module name, for example `AddressBookW2JE.jar`
- *EJB_name* is the name of the EJB if the module is not a Web module, for example Exchange
- *web_service* is the name of the Web service, for example service/StockQuoteService
- *deployed_WSDL_filename* identifies the WSDL file, relative to this module, for example, `META-INF/wsdl/AlternativeStockQuoteFetcher.wsdl`

3. Save the configuration changes with the **$AdminConfig save** command:

Results

Your Web service client bindings are configured.

Example

The following example presents the application, module and deployed WSDL file name as it is written in the command line:

```
$AdminApp edit MultiEjbJar {-WebServicesClientBindDeployedWSDL {{ejbclientonly.jar Exchange
service/StockQuoteService META-INF/wsdl/AlternateStockQuoteFetcher.wsdl}...}}
```

What to do next

Now you can finish any other configurations, start or restart the application, and verify the expected behavior of the Web service.

Configuring Web service client-preferred port mappings with the wsadmin tool

A client port type can be configured with ports that have different qualities of service. You can use the **WebServicesClientBindPreferredPort** command-line option to specify which port you want to use.

Before you begin

If you have not deployed the enterprise archive (EAR) file yet, you need to have it ready or already deployed to the application server.

About this task

For each port type that is configured, one or more ports are available that implement that port type. When a Web service client calls a getPort method, the preferred port mapping determines which port to use. This determination occurs when more than one port can satisfy the getPort method call, such as, a getPort call that specifies the port type, but not the port. For example, suppose the Web service client is configured to use both Java Message Service (JMS) and an HTTP implementation. During installation or management, you can use the **WebServicesClientBindPreferredPort** command to configure the preferred port of the application to use the transport of choice.

To configure the preferred port mapping with the wsadmin tool proceed with the following steps:

1. Launch a scripting command.
2. Configure Web service client-preferred port mappings.

To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.

Use the install, installInteractive, edit or editInteractive options to configure the Web service client-preferred port mappings, as the following syntax demonstrates:

- Using Jython:

```
AdminApp.install('app_name', '[-usedefaultbindings -deployejb
-WebServicesClientBindPreferredPort {{module_name EJB_name Web_service port_type
port_name}]')
```

- Using Jacl:

```
$AdminApp install app_name {-usedefaultbindings -deployejb
-WebServicesClientBindPreferredPort {{module_name EJB_name Web_service port_type
port_name}}
```

The example shows multiple modules and URL endpoints, because you can edit multiple URL fragments where:

- *app_name* is the application name, for example `MultiEjbJar.ear`
- *EJB_name* is the name of the enterprise bean module that is not a Web module, for example, Exchange
- *module_name* is the module name, for example `ejbclientonly.jar`
- *Web_service* is the name of the Web service, for example `service/StockQuoteService`
- *port_type* is the port type information, for example `{http://stock.multiejbjar.test.wsfvt.ws.ibm.com}StockQuote`
- *port_name* is the port name, for example `{http://stock.multiejbjar.test.wsfvt.ws.ibm.com}StockQuote`

Results

You have configured Web service client-preferred port mappings with the wsadmin tool.

Example

The following example includes the application, module, Web service, port type and port information as it is written in the command line:

```
$AdminApp install MultiEjbJar.ear {-WebServicesClientBindPreferredPort {{ejbclientonly.jar
Exchange service/StockQuoteService {http://stock.multiejbjar.test.wsfvt.ws.ibm.com}StockQuote
{http://stock.multiejbjar.test.wsfvt.ws.ibm.com}StockQuote}...}}
```

The port type information that drives the creation of the WebServicesClientBindPreferredPort option data resides in the client WSDL file. Because valid preferred port mappings are restricted to ports that implement the interface of the port type, validation requires the implementation type of each port. The client WSDL file must be accessed to determine both the type and the implementation information.

The client WSDL file name is in the ServiceRef attribute of the Web service client deployment descriptor. Depending on the module type and version, the client deployment descriptor is located in either the `application-client.xml` file; the `web.xml` file, or the `ejb-jar.xml` file. If you are using J2EE 1.3, the client deployment descriptor information is located in the `webservices.xml` file.

What to do next

Now you can finish any other configurations, start or restart the application, and verify expected behavior of the Web service.

Configuring Web service client port information with the wsadmin tool

A Web service can have multiple ports. You can view and configure the port attributes for each defined Web service port.

Before you begin

If you have not deployed the enterprise archive (EAR) file yet, you need to have it ready or already deployed to the application server.

About this task

This task supports configuring binding attributes that are associated with the Web service client port, including synchronization timeout, overridden endpoint URL and transport attributes with the **WebServicesClientBindPortInfo** command-line option. A typical usage scenario for this command-line option is to customize the timeout value of the client so that the client waits longer when it is configured to use a Java Message Service (JMS) transport to access a Web service.

1. Launch a scripting command.
2. Configure the Web service client port information.

   To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.

   Use install, installInteractive, edit or editInteractive options to configure the Web service client port information, as the following example demonstrates:

   ```
   $AdminApp install app_name {-usedefaultbindings
   -deployejb -WebServicesClientBindPortInfo {{module_name EJB_name
   Web_service port timeout basic_authentication_id basic_authentication_password
   SSL_alias overridden_endpoint overridden_binding_namespace }...}}
   ```

   The previous example indicates that the port information of multiple ports can be changed using one WebServicesClientBindPortInfo command, where:

   - *app_name* is the application name, for example, `MultiEjbJar.ear`
   - *module_name* is the module name, for example `ejbclientonly.jar`
   - *EJB_name* is the name of the EJB that is not a Web module, for example, Exchange
   - *Web_service* is the name of the Web service, for example service/StockQuoteService
   - *port* is the name of the port, for example StockQuote
   - *timeout* specifies the number of seconds that the client waits for a response
   - *basic_authentication_id* is the basic authentication transport ID
   - *basic_authentication_password* is the basic authentication transport password
   - *SSL_alias* identifies the SSL alias for the port
   - *overridden_endpoint* is the name of the endpoint that is used to override the current endpoint
   - *overridden_binding_namespace* specifies the WSDL file binding namespace URI to use with the port
3. Save the configuration changes with the **$AdminConfig save** command:

Results

The client port information that is associated with the Web service client port are configured.

Example

```
$AdminApp installInteractive MultiEjbJar.ear {-WebServicesClientBindPortInfo
 {{ejbclientonly.jar Exchange service/StockQuoteService StockQuote 6000 jsmith js9password level3ssl
 http://fastball.houston.ibm.com/newURL http://fastball.houston.ibm.com/newBindName}}
 {ejbclientonly.jar Exchange service/StockQuoteService StockQuote2 9000 {}{}{}{}{}}}
```

What to do next

Now you can finish any other configurations, start or restart the application, and verify expected behavior of the Web service.

Configuring the scope of a Web service port with the wsadmin tool

When a Web service application is deployed into WebSphere Application Server, an instance is created for each application or module. The instance contains deployment information for the Web module or Enterprise JavaBeans (EJB) module, including implementation scope, client bindings and deployment descriptor information. There are three levels of scope that can be set: application, session and request.

Before you begin

If you have not deployed the enterprise archive (EAR) file yet, you need to have it ready or already deployed to the application server.

About this task

The primary purpose of this task is to enable the configuration of the Web service port scope. The scope originally specified when the JavaBean object is enabled as a Web service during the development process can be changed with the **WebServicesServerBindPort** command. The scope attribute does not apply to Web services that use Java Message Service (JMS) transport or to enterprise beans.

Web Services for Java 2 platform Enterprise Edition (J2EE) specifies that Web services implementations must be stateless. Therefore, to maintain specification compliance, the scope can remain at the application level because the state relevant to the individual sessions level or the requests level is not supposed to be maintained in the implementation. If you want to deviate from the specification and want to access a different JavaBean instance, because you are looking for information that is located in another JavaBean, the scope settings need to change.

The setting that you configure for the scope determines how frequently a new instance of a service implementation class is created for the Web service ports in a module. The application scope causes the same instance of the implementation to be used for all requests on the application. The session scope causes the same instance to be used for all requests in each session. The request scope causes a new instance to be used for every request. For example, with the scope set to application, every message that comes to the server accesses the same Java bean instance.

To change the scope setting through the wsadmin tool:

1. Launch a scripting command.
2. Configure the scope of the Web service port.

   To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the

defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.

Use the install, installInteractive, edit, or editInteractive options to configure the scope of the Web service port, as the following syntax demonstrates:

```
$AdminApp install app_name {-usedefaultbindings -deployejb
-WebServicesServerBindPort {{<module_name> <Web_service> <port><scope_setting>}...}
```

The previous example indicates that the scope of multiple ports can be changed using one WebServicesServerBindPort command, where:

- *app_name* is the application name, for example `WebServicesSample.ear`
- *module_name* is the module name, for example `AddressBookW2JE.jar`
- *Web_service* is the name of the Web service, for example `AddressBookW2JE service/ WSLoggerService2`
- *port* is the name of the port, for example `AddressBook`
- *scope_setting* is the level of setting for scope, for example `Session`

Results

The scope for a Web service port is configured.

Example

The following example of the presents the application, module, Web service, port and scope as it is written in the command line:

```
$AdminApp install WebServicesSamples.ear {-usedefaultbindings -deployejb -deployws
-WebServicesServerBindPort {{AddressBookJ2WB.war AddressBookService AddressBook request}
{AddressBookW2JB.war AddressBookService AddressBook application}}}
```

What to do next

Now you can finish any other configurations, start or restart the application, and verify expected behavior of the Web service.

Publishing WSDL files using the wsadmin tool

The Web Services Description Language (WSDL) files in each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files in the development and configuration process of Web services clients so that they can invoke your Web services.

Before you begin

Before you publish a WSDL file, you can configure Web services to specify endpoint information in the form of URL fragments to enable full URL specification of WSDL ports. Refer to the tasks describing configuring endpoint URL information.

To publish a Web Services Description Language (WSDL) file you need an enterprise application, also known as an enterprise archive (EAR) file, that contains a Web services-enabled module and has been deployed into WebSphere Application Server. See Deploying Web services based on Web Services for Java 2 platform, Enterprise Edition (J2EE).

About this task

The purpose of publishing the WSDL file is to provide clients with a description of the Web service, including the URL identifying the location of the service.

After installing a Web services application, and optionally modifying the endpoint information, you might need WSDL files containing the updated endpoint information. You can obtain the updated WSDL files by publishing them to the file system. If you are a client developer or a system administrator, you can use WSDL files to enable clients to connect to a Web service.

The wsadmin tool can publish the WSDL files in either local, for example, -conntype NONE, or remote mode. However, in local mode, locate the target application at the same node where the **wsadmin** command is invoked.

The following steps assume that the application has been deployed and that the application server is running.

1. Start the wsadmin tool from the command prompt using the following commands:
2. At the wsadmin command prompt, enter one of the two commands:
   - **$AdminApp publishWSDL***app_name path_name*
   - **$AdminApp publishWSDL***app_name path_name soapAddressPrefixes*

   where:
   - *app_name* is the application name
   - *path_name* is the absolute path to the zip file that contains the published WSDL files. The zip file is saved on the machine that runs WebSphere Application Server, therefore, if the server is running on a different machine, you need to obtain the zip file from that machine. The directory structure of the resulting zip file is based on the following information:

     `Application_file_name/module_file_name/META-INF/ or WEB-INF/wsdl/WSDL_file_name`

     See the usage scenario for an example of this directory structure.
   - *soapAddressPrefixes* is a parameter of the form {{*module* {{*binding partial-url*}}}}. This parameter describes the partial URL information for each binding on a per-module basis for the application.
     - *module* identifies the name of the module
     - *binding* is either http or jms (both are in lower case)
     - *partial-url* is the partial SOAP address for the associated SOAP binding. For an HTTP binding, the form is `http://host:port/` or `https://host:port`.

       For Java Message Service (JMS) bindings, the form is

       `jms:/queue?destination=`*dest*`&connectionFactory=cf`

       or

       `jms:/topic?destination=`*dest*`&connectionFactory=cf`

   Use the **$AdminApp publishWSDL *app_Name path_Name*** command to publish WSDL files with default endpoint URL addresses. If you want to modify the SOAP address prefixes of the WSDL file, use the other form of the command.

   Use the **$AdminApp publishWSDL *app_Name path_Name* {{*module* {{*binding partial-url*}}}}** command to customize the WSDL SOAP address for each module. You can specify a different address prefix for each SOAP binding.

Results

The WSDL files from Web services are published to a specified zip file. The zip file can be used to create a Web service client that accesses the deployed service. The published WSDL files do not contain enterprise JavaBean (EJB) binding information.

Example

The command to publish WSDL files for a Web service named WebServicesSamples can be **$AdminApp publishWSDL WebServicesSamples c:/temp/samplesWsdl.zip**

or

```
$AdminApp publishWSDL WebServicesSamples c:/temp/sampleswsdl.zip { {AddressBookJ2WB.war {{http
http://localhost:9080}}} {StockQuote.jar {{http https://localhost:9443}}} }
```

The directory structure for this created zip file is

```
WebServicesSamples.ear/StockQuote.jar/META-INF/wsdl/StockQuoteFetcher.wsdl
WebServicesSamples.ear/AddressBookW2JE.jar/META-INF/wsdl/AddressBookW2JE.wsdl
WebServicesSamples.ear/AddressBookJ2WE.jar/META-INF/wsdl/AddressBookJ2WE.wsdl
WebServicesSamples.ear/AddressBookJ2WB.war/WEB-INF/wsdl/AddressBookJ2WB.wsdl
WebServicesSamples.ear/AddressBookW2JB.war/WEB-INF/wsdl/AddressBookW2JB.wsdl
```

What to do next

Develop a Web services client or configure the endpoint information for an existing Web service.

Configuring application and system policy sets for Web services using scripting

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

Before you begin

Develop a Web services application. For additional information, see the Web services applications topics in the information center.

If you develop an application that uses a custom policy set, the custom policy set configuration is not included in the application enterprise archive (EAR) file. Install the application and import the custom policy set separately.

About this task

The commands in the PolicySetManagement group for the AdminTask object configure both application and system policy sets. Use the following tasks to configure and manage policy sets for your Web services.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Use the following guidelines to manage bindings in your environment:

- To display or modify default Version 6.1 bindings, Version 7.0 trust service bindings, or to reference bindings by attachment for an application, specify the attachmentId and bindingLocation parameters with the getBinding or setBinding commands.
- To use or modify general Version 7.0 bindings, specify the bindingName parameter with the getBinding or setBinding commands.
- To display the version of a specific binding, specify the **version** attribute for the getBinding command.

Use a Version 6.1 binding for an application in a Version 7.0 environment if:

- The module in the application is installed on at least one Web Services Feature Pack server.
- The application contains at least one Version 6.1 application-specific binding. The application server does not assign general bindings to resource attachments for applications that are installed on a Web Services Feature Pack server. All application-specific bindings for an application must be at the same level.

General service provider and client bindings are not linked to a particular policy set and they provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that are deployed to a server to share binding configuration. You can also accomplish this sharing of binding configuration by assigning the binding to each application deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:

1. Server level default
2. Security domain level default
3. Global security (cell) default

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding.

- Use the PolicySetManagement group of commands to configure application and client policy sets:
  - Create a new policy set or copy an existing policy set.
  - Add policies to your policy set.
  - Attach your policy set to an application, Web service, endpoint, or operation.
  - Customize cell-wide, server-specific, or application binding configurations.
  - Manage and edit your policy set configurations.
    - Edit, enable, disable, or remove policies.
    - Add, edit, or remove policy set attachments.
    - Export and import policy sets.
    - Delete policy sets.
- Use the PolicySetManagement group of commands to configure system policy sets.
  - Create a new system policy set or copy an existing system policy set.
  - Add policy types for your policy set.
  - Add trust service attachments.

– Customize binding configurations.
– Manage and edit your policy set configurations.
  - Edit, enable, disable or remove policies.
  - Add, edit, or remove policy set attachments.
  - Export and import policy sets.
  - Delete policy sets.

Creating policy sets using the wsadmin tool

Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

Before you begin

In order to complete this task, you must use the Administrator role with cell-wide access when administrative security is enabled.

About this task

There are three ways to create a new policy set using the wsadmin tool. You can create a new policy set and its configuration, copy an existing policy set, or import a policy set.

When you create a new policy set, you must add policies. If you copy an existing policy set, you can transfer the policies and attachments that are configured on the existing policy set. The command examples in this topic use batch mode syntax. You can use the -interactive option with all commands in the PolicySetManagement group.

- Create a new policy set using the Jython scripting language.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the policy requirements for your Web services.
  3. Enter the command syntax to create a new policy set with a given name.

     Based on your configuration, there are two types of policy sets to create. You can use both application and system policy sets with Java API for XML-Based Web Services (JAX-WS) applications. Use the -policySetType parameter to specify the type of policy set. To create an application policy set, specify `application` for the value of the -policySetType parameter. To create a policy set for the trust service, specify `system` or `system/trust` for the -policySetType parameter. For WS-MetadataExchange attachments, specify `system` for the -policySetType parameter. The -policySetType parameter is optional. The wsadmin tool creates an application policy set if the -policySetType parameter is not specified.

     Enter the following command to create an application policy set:

```
AdminTask.createPolicySet('[-policySet PolicySet1 -description policySet_description]')
```

     Enter the following command to create a policy set for the trust service:

```
AdminTask.createPolicySet('[-policySet PolicySet1 -description policySet_description -policySetType system]')
```

     The command returns a success or failure message.
  4. Add policies for your new policy set. Use this step to add a policy with default values for the specified policy set.

     Enter the following command to add and enable a policy:

```
AdminTask.addPolicyType('[-policySet PolicySet1 -policyType policyType_name]')
```

     Enter the following command to add and disable a policy. Your configuration changes are contained within the policy set, but will have no effect on the system if the -enabled parameter is set to `false`.

```
AdminTask.addPolicyType('[-policySet PolicySet1 -policyType policyType_name -enabled false]')
```

The command returns a success or failure message. Repeat this step to create additional policies for your configuration.

5. Save the configuration changes.

   Enter the following command to save your changes:

```
AdminConfig.save()
```

- Copy an existing policy set using the Jython scripting language.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the policy requirements for your Web services.
  3. Enter the command syntax to copy an existing policy set:

     Set the -transferAttachments parameter to `true` to transfer the attachments from the existing policy set to the new policy set. The default value for the -transferAttachments parameter is `false`.

     Enter the following command to create the new policy set and to transfer the attachments of the existing policy set:

```
AdminTask.copyPolicySet('[-sourcePolicySet existingPolicySet_name -newPolicySet PolicySet1 -newDescription PolicySet1_description -transferAttachments true]')
```

     The command returns a success or failure message.

  4. Save the configuration changes.

     Enter the following command to save your changes:

```
AdminConfig.save()
```

- Import a policy set from an archive file or import a default policy set using the Jython scripting language.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the policy requirements for your Web services.
  3. Import a policy set.

     Use the importPolicySet command to import the archive file containing the policy set configuration of interest to the destination environment. Specify the verifyPolicySetType parameter to verify that the policy set to import matches a specific type. Set the value as `application`, `system`, or `system/trust` to specify the policy set type. You cannot import a policy set onto a server or client environment if the policy set already exists in the destination environment.

     For example, the following command creates a customSecureConversation policy set from the customSC.zip archive file:

```
AdminTask.importPolicySet('[-importFile /IBM/WebSphere/AppServer/bin/customSC.zip -verifyPolicySetType system/trust]')
```

     Additionally, you can also use the importPolicySet command to import a default policy set onto a server environment, as the following example demonstrates:

```
AdminTask.importPolicySet('[-defaultPolicySet SecureConversation -policySet copyOfdefaultSC -verifyPolicySetType system]')
```

     The command returns a success or failure message.

  4. Save the configuration changes.

     Enter the following command to save your changes:

```
AdminConfig.save()
```

Results

If you receive a success message after entering the commands, you can now manage a policy set that is customized for your Web services applications. You can further configure the policy set and policies.

What to do next

Use the **validatePolicySet** command to validate your policy set configurations after modifying attributes for policies. For example, enter the following command to validate the *PolicySet1* policy set:

```
AdminTask.validatePolicySet('-policySet PolicySet1')
```

Related concepts

Web services policy sets
Policy sets are assertions about how services are defined. They are used to simplify your quality of service configuration for Web services.

Related tasks

"Updating policy set attributes using the wsadmin tool"
Use policy sets to centrally manage policies that are customized for your Web services. Use the Jython or Jacl scripting language with the wsadmin tool to update policy set attributes. You can also query the configuration for an existing policy set and respective attributes.

"Deleting policy sets using the wsadmin tool" on page 1080
Use the Jython or Jacl scripting language to delete policy sets from your configuration with the wsadmin tool. You must remove all policy set attachments before removing the policy set.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

Creating policy sets using the administrative console
You can use the administrative console to either create a policy set by specifying all the necessary information or by copying an existing policy set that you rename. You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Updating policy set attributes using the wsadmin tool

Use policy sets to centrally manage policies that are customized for your Web services. Use the Jython or Jacl scripting language with the wsadmin tool to update policy set attributes. You can also query the configuration for an existing policy set and respective attributes.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to update policy sets.
Configurator	The Configurator role cannot update policy sets.
Deployer	The Deployer role cannot update policy sets.
Operator	The Operator role cannot update policy sets.
Monitor	The Monitor role cannot update policy sets.

About this task

After creating a new policy set, you can update your policy set configuration with the updatePolicySet command. The command uses a properties object to update all attributes or a subset of attributes for the specified policy set.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. View the current attribute values for the policy set of interest.

   Enter the following command to display the attribute values of the *policySet1* policy set:

```
AdminTask.getPolicySet('[-policySet policySet1]')
```

3. Modify attributes for the policy set of interest.

   Enter the following command to modify the `type` and `description` attributes for the *policySet1* policy set.

```
AdminTask.updatePolicySet('-policySet policySet1 -attributes "[ [type application] [description [my custom policy set to manage WSSecurity]] ]"')
```

   You can also use the -interactive option to update a policy set. To start -interactive mode, enter this command:

```
AdminTask.updatePolicySet('-interactive')
```

4. Save the configuration changes.

   Enter the following command to save your changes:

```
AdminConfig.save()
```

Results

If you received a success message after entering the commands, manage and customize the policy set for your Web services applications.

What to do next

After updating policy set attributes, you can further configure policies and policy set attachments using the commands and parameters for the PolicySetManagement group of the AdminTask object.

Related concepts

Web services policy sets
Policy sets are assertions about how services are defined. They are used to simplify your quality of service configuration for Web services.

Related tasks

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Deleting policy sets using the wsadmin tool" on page 1080
Use the Jython or Jacl scripting language to delete policy sets from your configuration with the wsadmin tool. You must remove all policy set attachments before removing the policy set.

"Adding and removing policies using the wsadmin tool"
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

Creating policy sets using the administrative console
You can use the administrative console to either create a policy set by specifying all the necessary information or by copying an existing policy set that you rename. You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units.

"Policy configuration properties for all policies" on page 1083
You can use the **attributes** parameter with the setPolicyType and setBinding commands to specify various properties for each quality of service (QoS) within a policy set. You can use the properties in this topic with each QoS within application and system policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Adding and removing policies using the wsadmin tool

You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

Additionally, if administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to create and remove policies.
Configurator	The Configurator role cannot create or remove policies.
Deployer	The Deployer role cannot create or remove policies.
Operator	The Operator role cannot create or remove policies.
Monitor	The Monitor role cannot create or remove policies.

About this task

Policies define which Qualities of Service (QoS) to manage within a policy set. Policy definitions are based on the standards set by the Organization for the Advancement of Structured Information (OASIS) and Web Services Security specifications.

For application policy sets, you can add the following policies:

- WSSecurity
- WSReliableMessaging
- WSAddressing
- HTTPTransport
- SSLTransport
- WSTransaction
- JMSTransport

For system policy sets, you can add the following policies:

- WSSecurity
- WSAddressing
- HTTPTransport
- SSLTransport
- WS-MetadataExchange

Use the following steps to add or remove policy types from your policy set configurations:

- Add a policy to a policy set. Use this section to add a policy with default values to the specified policy set. You can create and enable or create and disable the policy.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. List all policies for a specified policy set.

     Enter the following command and specify the policy set of interest to list all policies that have been added to the policy set:

     ```
     AdminTask.listPolicyTypes('[-policySet PolicySet1]')
     ```

     Enter the following command to list all the available policies:

     ```
     AdminTask.listPolicyTypes()
     ```
  3. Add the policy to your configuration.

     Enter the following command to add and enable a policy:

     ```
     AdminTask.addPolicyType('[-policySet PolicySet1
     -policyType policyType_name]')
     ```

     Enter the following command to add and disable a policy. Your configuration changes are contained within the policy set, these changes do not effect the system if the -enabled parameter is set to `false`.

     ```
     AdminTask.addPolicyType('[-policySet PolicySet1
     -policyType policyType_name -enabled false]')
     ```
  4. Enter the following command to save your changes:

     ```
     AdminConfig.save()
     ```
  5. For your configuration changes to take effect, restart all applications with attachments to the policy set.

  The command returns a success or failure message. Repeat this step to create additional policies for your configuration.
- Remove a policy from the policy set configuration. The deletePolicyType command removes the specified policy from the policy set. Applications with attachments to the policy set are not affected until the application restarts.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Enter the following command to list all policies for the policy set of interest:

   ```
   AdminTask.listPolicyTypes('[-policySet PolicySet1]')
   ```

3. Enter the following command to remove the policy:

   ```
   AdminTask.deletePolicyType('[-policySet PolicySet1
   -policyType policyType_name]')
   ```

   The command returns a success or failure message.
4. Save the configuration changes.

   Enter the following command to save your changes:

   ```
   AdminConfig.save()
   ```

5. For your configuration changes to take effect, restart all applications with attachments to the policy set.

What to do next

Use the validatePolicySet command to validate your policy set configurations after modifying attributes for policies. For example, enter the following command to validate the *PolicySet1* policy set:

```
AdminTask.validatePolicySet('-policySet PolicySet1')
```

Related concepts

Web services policies
Policies define the type of Web service policy based on the quality of service type. Policies are initially set with default settings but the attributes can be edited and changed.

Related tasks

"Editing policy configurations using the wsadmin tool"
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to edit policy configurations for your policy sets.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Enabling policies for policy sets using the administrative console
Policies can be listed in a policy set in the disabled state so that they are not currently included in the policy set. You can enable a policy to be included in a policy set using the administrative console.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

Adding policies to policy sets using the administrative console
You can use the administrative console to add policies to policy sets. Adding and configuring policies to a policy set further defines the rules governing the policy set.

Managing policies in a policy set using the administrative console
When working with policy sets in the administrative console, you can customize the included policies to ensure message security. You can enable, disable, customize, add, or delete policies from a policy set. With your policy sets, you can define policies for WS-Addressing, WS-Security, WS-ReliableMessaging, WS-Transaction, HTTP transport, Java Messaging Service (JMS) transport, and Secure Sockets Layer (SSL) transport. The policies for all but WS-Security are relatively straightforward to define.

Related reference

"Policy configuration properties for all policies" on page 1083
You can use the **attributes** parameter with the setPolicyType and setBinding commands to specify various properties for each quality of service (QoS) within a policy set. You can use the properties in this topic with each QoS within application and system policy sets.

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Editing policy configurations using the wsadmin tool

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to edit policy configurations for your policy sets.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to modify policies.
Configurator	The Configurator role cannot modify policies.
Deployer	The Deployer role cannot modify policies.
Operator	The Operator role cannot modify policies.
Monitor	The Monitor role cannot modify policies.

About this task

Policies define the type of policy to manage within a policy set. Policies are based on the Quality of Services (QoS), such as Web Services Security (WS-Security) and Web Services Addressing (WS-Addressing). Policy definitions are based on the standards set by the Organization for the Advancement of Structured Information (OASIS) and WS-Security specifications.

Use the following steps to edit existing policies in your policy set configurations:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine which policy set to edit.

   To view a list of policies on a policy set, enter the listPolicyTypes command, specifying the policy set of interest.

```
AdminTask.listPolicyTypes('[-policySet PolicySet1]')
```

   Enter the listPolicyTypes command without the policySet parameter to view a list of available policies for all policy sets in your configuration:

```
AdminTask.listPolicyTypes()
```

3. Review the policy attributes to edit.

   Enter the getPolicyType command, specifying the policy and associated policy set of interest.

```
AdminTask.getPolicyType('[-policySet PolicySet1 -policyType myPolicyType]')
```

4. Modify the policy set attributes.

   Use the setPolicyType command to update the policy configuration. Update one or multiple attributes by passing a properties object for the -attributes parameter. The following example modifies the `enabled` and `provides` properties:

```
AdminTask.setPolicyType('[-policySet PolicySet1 -policyType myPolicyType
 -attributes "[[enabled true][provides security]]"')
```

5. Save the configuration changes.

   Enter the following command to save your changes:

```
AdminConfig.save()
```

6. For your configuration changes to take effect, restart all applications with attachments to the policy set.

What to do next

Use the validatePolicySet to validate your policy set configurations after modifying attributes for policies. For example, enter the following command to validate the *PolicySet1* policy set:

```
AdminTask.validatePolicySet('-policySet PolicySet1')
```

Related concepts

Web services policies
Policies define the type of Web service policy based on the quality of service type. Policies are initially set with default settings but the attributes can be edited and changed.

Related tasks

Modifying policies using the administrative console
With your policy sets, you can define policies for WS-Addressing, WS-Security, WS-ReliableMessaging, WS-Transaction, HTTP transport, Java Messaging Service (JMS) transport, and Secure Sockets Layer (SSL) transport. The policies for all but WS-Security are relatively straightforward to define.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Enabling policies for policy sets using the administrative console
Policies can be listed in a policy set in the disabled state so that they are not currently included in the policy set. You can enable a policy to be included in a policy set using the administrative console.

Adding policies to policy sets using the administrative console
You can use the administrative console to add policies to policy sets. Adding and configuring policies to a policy set further defines the rules governing the policy set.

Managing policies in a policy set using the administrative console
When working with policy sets in the administrative console, you can customize the included policies to ensure message security. You can enable, disable, customize, add, or delete policies from a policy set. With your policy sets, you can define policies for WS-Addressing, WS-Security, WS-ReliableMessaging, WS-Transaction, HTTP transport, Java Messaging Service (JMS) transport, and Secure Sockets Layer (SSL) transport. The policies for all but WS-Security are relatively straightforward to define.

Related reference

"Policy configuration properties for all policies" on page 1083
You can use the **attributes** parameter with the setPolicyType and setBinding commands to specify various properties for each quality of service (QoS) within a policy set. You can use the properties in this topic with each QoS within application and system policy sets.

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Enabling secure conversation using the wsadmin tool

Use this topic and the commands in the SecureConversation group of the AdminTask object to enable secure conversation client cache by creating a new policy set and bindings to attach to your applications.

Before you begin

Verify that the SecureConversation policy set is available in your configuration. By default, the SecureConversation policy set is not available. Use the importPolicySet command to import the SecureConversation policy to your configuration, as the following example demonstrates:

```
AdminTask.importPolicySet('-defaultPolicySet SecureConversation')
```

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

About this task

This topic uses the default SecureConversation policy set and default WS-Security and TrustServiceSecurityDefault bindings to enable secure conversation.

The default SecureConversation policy set contains an application policy with the symmetric binding, and a bootstrap policy with the asymmetric binding. The application policy secures application messages. The bootstrap policy secures RequestSecurityToken (RST) messages. The trust service, which issues security context token providers, uses the TrustServiceSecurityDefault system policy and the TrustServiceSecurityDefault bindings. The trust policy secures RequestSecurityTokenResponse (RSTR) messages. If you modify the bootstrap policy, you must also modify the trust policy so that both of the configurations match.

Note: Use the following steps in development and test environments only. The WS-Security bindings in this procedure contain sample key files that you must customize before using the bindings in a production environment. Create custom bindings for your production environment.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Copy the existing SecureConversation policy set.

   Use the following command example to create a new policy set by copying the existing SecureConversation policy set:

```
AdminTask.copyPolicySet('[-sourcePolicySet SecureConversation -newPolicySet CopyOfSCPolicySet]')
```

3. Change the binding for the global security domain. If you chose the **Create the server using the development template** option when you created your profile with the Profile Management Tool or the manageprofiles command utility, you can optionally skip this step.
   a. List each WS-Security policy attribute.

      To modify the binding for the global security domain, use the getDefaultBindings command to determine the binding that is set as the default for the provider or client, as the following example demonstrates:

```
AdminTask.getDefaultBinding('-bindingType provider')
```

   b. Display the attributes for the binding.

      Use the getBinding command to display the current attributes for the binding, as the following example demonstrates:

```
AdminTask.getBinding('-bindingLocation "" -bindingName myBinding')
```

   c. Modify the outbound configuration for the protection token.

      Use the following commands to modify the outbound configuration for the protection token:

```
cmd1_attributes_value = "[ [application.securityoutboundbindingconfig.tokengenerator_5.callbackhandler
.key.name [CN=Bob,O=IBM, C=US]] [application.securityoutboundbindingconfig.tokengenerator_5.callbackhandler
.keystore.storepass storepass] [application.securityoutboundbindingconfig.tokengenerator_5.callbackhandler
.keystore.type JCEKS] [application.securityoutboundbindingconfig.tokengenerator_5.callbackhandler.key.alias bob]
[application.securityoutboundbindingconfig.tokengenerator_5.callbackhandler.keystore.path ${USER_INSTALL_ROOT}
/etc/ws-security/samples/enc-sender.jceks] ]"

AdminTask.setBinding('[-policyType WSSecurity -bindingLocation "" -attributes cmd1_attributes_value
-attachmentType application]')

cmd2_attributes_value = "[ [application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore
.path ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks] [application.securityoutboundbindingconfig
.tokengenerator_0.callbackhandler.keystore.storepass client] [application.securityoutboundbindingconfig
.tokengenerator_0.callbackhandler.key.name [CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP]] [application
.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key.keypass client] [application
.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key.alias soaprequester]
```

```
[application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.type JKS] ]"

AdminTask.setBinding('[-policyType WSSecurity -bindingLocation "" -attributes cmd2_attributes_value
-attachmentType application]')
```

4. Optional: Modify the TrustDefaultBindings binding. If you chose the **Create the server using the development template** option when you created your profile with the Profile Management Tool or the manageprofiles command utility, you can optionally skip this step.

   If the TrustDefaultBindings are not yet customized, use the following commands to modify the TrustDefaultBindings binding:

```
cmd3_attributes_value = "[  [application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.keystore
.storepass storepass] [application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.key.alias bob]
[application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.keystore.type JCEKS] [application
.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.keystore.path ${USER_INSTALL_ROOT}/etc
/ws-security/samples/enc-sender.jceks] [application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler
.key.name [CN=Bob, O=IBM, C=US]] ]"

AdminTask.setBinding('[-policyType WSSecurity -bindingLocation "[attachmentId 2]"
-attributes cmd3_attributes_value -attachmentType system/trust]')

cmd4_attributes_value = "[ [application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.path
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks] [application.securityoutboundbindingconfig.tokengenerator_0
.callbackhandler.keystore.storepass client] [application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler
.key.name [CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP]] [application.securityoutboundbindingconfig.tokengenerator_0
.callbackhandler.key.keypass client] [application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key
.alias soaprequester] [application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.type JKS] ]"

AdminTask.setBinding('[-policyType WSSecurity -bindingLocation "[attachmentId 2]"
-attributes cmd4_attributes_value -attachmentType system/trust]')
```

5. Attach the policy set and binding to the application.

   Use the attachmentType parameter for the createPolicySetAttachment command to specify if your application is a service client or a service provider. Use the following commands to attach the *CopyOfSCPolicySet* policy set to the *myTestApp* service client application:

```
AdminTask.createPolicySetAttachment('[-applicationName myTestApp -policySet CopyOfSCPolicySet
-resources WebService:/ -attachmentType client]')
```

   Use the following commands to attach the *CopyOfSCPolicySet* policy set to the *myTestApp* service provider application:

```
AdminTask.createPolicySetAttachment('[-applicationName myTestApp -policySet CopyOfSCPolicySet
-resources WebService:/ -attachmentType application]')
```

   This step automatically assigns the bindings.

Results

Your secure conversation configuration is updated in the WSSCCache.xml file located in the cell level directory.

What to do next

Manage your secure conversation configurations with the SecureConversation command group for the AdminTask object.

Related concepts

Secure conversation client cache
For both distributed and local clients, the WebSphere Application Server secure conversation client cache stores tokens on the client.

SecureConversation default policy sets
The SecureConversation default policy sets are based on the Web Services Secure Conversation Language (SecureConversation) standard that establishes a secure context, based on shared keys for the client and server to use for a series of messages. This standard provides a framework to define how to secure the message exchange across organizations. The SecureConversation default policy sets include the SecureConversation policy set, the Lightweight Third-Party Authentication (LTPA) SecureConversation policy set, and the Username SecureConversation policy set.

Related tasks

Configuring the Web services security distributed cache using the administrative console
You can configure the Web services security runtime to use the security distributed cache to store security tokens.

Example: Installing a Web Services Sample with the console
The product provides a Web Services sample application that you can install on a Version 7.x application server.

Managing WS-Security distributed cache configurations using the wsadmin tool

The distributed cache stores tokens on the client. Use this topic and the commands in the WSSCacheManagement group of the AdminTask object to query, update, and remove custom and non-custom properties for the distributed cache configuration.

Before you begin

Configure a policy set with WS-Security enabled.

About this task

The distributed cache stores tokens on both distributed and local clients. WebSphere Application Server supports only the security context token for the WS-Trust security token service client and the security trust service components.

You can use the administrative console or the wsadmin tool to manage your secure conversation distributed cache configuration. You can use the wsadmin tool and the Jython scripting language syntax to:

- Query your current distributed cache configuration settings.
- Set the value for the renewal time after token expiration.
- Enable or disable distributed cache for clustered servers.
- Add custom properties to your configuration.
- Remove custom properties from your configuration.
- Query the configuration for your existing distributed cache configuration.

  You can retrieve a list of your current distributed cache configuration settings and custom properties with the queryWSSDistributedCacheConfig and queryWSSDistributedCacheCustomConfig commands. There are no required or optional parameters for the query commands.

  To list all non-custom configuration settings, run the following Jython command:

```
AdminTask.queryWSSDistributedCacheConfig()
```

  To list all distributed cache custom properties, enter the following Jython command:

```
AdminTask.queryWSSDistributedCacheCustomConfig()
```

- Update your secure conversation distributed cache configuration settings and custom properties.

Use the following steps to update all non-custom distributed cache configuration settings:

1. Review your existing configuration settings by running the queryWSSDistributedCacheConfig command, as the following example demonstrates:

```
AdminTask.queryWSSDistributedCacheConfig()
```

   The command returns a properties object that contains the configuration properties and values for the distributed cache configuration. The following table displays the configuration properties that the command returns:

Property	Description
tokenRecovery	Specifies whether token recovery is enabled or disabled. If the tokenRecovery property is set to `true`, the Datasource property specifies the shared data source that is assigned to the distributed cache.
distributedCache	Specifies whether distributed caching is enabled or disabled.
Datasource	Specifies the name of the shared data source that is assigned to the distributed cache if token recovery is enabled.
renewIntervalBeforeTimeoutMinutes	Specifies the amount of time, in minutes, that the client waits before it attempts to renew the token.
synchronousClusterUpdate	Specifies whether the system performs a synchronous update of distributed caches on cluster members. By default, synchronous cluster updating is enabled.
minutesInCacheAfterTimeout	Specifies the amount of time that the token remains in the cache after the token times out.

2. Use the updateWSSDistributedCacheConfig command to enable or disable distributed cache and to modify the amount of time after token expiration when downstream calls are allowed to complete.

   The following command example enables distributed cache, and sets the `mySharedDataSource` as the shared data source for token recovery:

```
AdminTask.updateWSSDistributedCacheConfig('[-tokenRecovery true -Datasource
 mySharedDataSource -distributedCache true')
```

3. Enter the following command to save your configuration changes:

```
AdminConfig.save()
```

Use the following steps to update custom properties for your distributed cache configuration:

1. Review your existing configuration settings by executing the queryWSSDistributedCacheCustomConfig command. For example:

```
AdminTask.queryWSSDistributedCacheCustomConfig()
```

   The command returns a properties object that contains the name and value pairs that correspond to each custom property.

2. Use the updateWSSDistributedCacheCustomConfig command to add custom properties for your distributed cache configuration. Specify and define each custom property by passing a properties object with the -customProperties parameter using the following Jython format:

```
-customProperties [[property1 value1][property2 value2]]
```

   For example, the following command adds the `cancelActionRST` custom property and defines the value as `http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT/Cancel`:

```
AdminTask.updateWSSDistributedCacheCustomConfig('[-customProperties [[cancelActionRST
 http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT/Cancel]]]')
```

3. Enter the following command to save your configuration changes:

```
AdminConfig.save()
```

- Remove custom properties from your distributed cache configuration. Use the following steps to remove custom properties from your distributed cache configuration:

  1. Review your existing configuration settings by executing the queryWSSDistributedCacheCustomConfig command. For example:

```
AdminTask.queryWSSDistributedCacheCustomConfig()
```

2. Use the deleteWSSDistributedCacheConfigCustomProperties command to remove custom properties for your distributed cache configuration. Specify the custom properties to delete by passing a string array with the -propertyNames parameter. For example, the following command removes the `cancelActionRST` custom property:

```
AdminTask.deleteWSSDistributedCacheConfigCustomProperties('[-propertyNames
 [cancelActionRST]]')
```

3. Enter the following command to save your configuration changes:

```
AdminConfig.save()
```

Results

Your WS-Security distributed cache configuration is updated.

Related concepts

Secure conversation client cache
For both distributed and local clients, the WebSphere Application Server secure conversation client cache stores tokens on the client.

SecureConversation default policy sets
The SecureConversation default policy sets are based on the Web Services Secure Conversation Language (SecureConversation) standard that establishes a secure context, based on shared keys for the client and server to use for a series of messages. This standard provides a framework to define how to secure the message exchange across organizations. The SecureConversation default policy sets include the SecureConversation policy set, the Lightweight Third-Party Authentication (LTPA) SecureConversation policy set, and the Username SecureConversation policy set.

Related tasks

Configuring the Web services security distributed cache using the administrative console
You can configure the Web services security runtime to use the security distributed cache to store security tokens.

Example: Installing a Web Services Sample with the console
The product provides a Web Services sample application that you can install on a Version 7.x application server.

Configuring custom policies and bindings for security tokens using the wsadmin tool

Use the setPolicyType and setBinding commands for the AdminTask object to specify security tokens for custom policy and binding configurations.

Before you begin

Create a new custom policy set.

About this task

The following scenarios configure the custom policy and bindings to use a Kerberos token based on the Oasis Kerberos Token Profile V1.1 specification. You can also use the setPolicyType and setBinding commands to configure other binary security tokens, such as username tokens, Lightweight Third-Party Authentication (LTPA) and SecureConversation.

- Configure custom policies for security tokens.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Display the properties of the policy of interest.

     Use the getPolicyType command to display detailed property information for the WS-Security policy type, as the following command demonstrates:

```
AdminTask.getPolicyType('-policySet AuthenticationTokenService -policyType
 WSSecurity')
```

The getPolicyType command returns a properties object that contains name and value pairs for each property, as the following sample output displays:

```
'[ [SupportingTokens.request:krb_token.CustomToken.IncludeToken
 http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient] [enabled true] [type WSSecurity]
 [description [Policies for sending security tokens and providing message confidentiality and integrity, based on the OASIS Web
 Service Security and Token Profiles specifications.]] [SupportingTokens.request:krb_token.CustomToken.WssCustomToken.uri ]
 [provides ] [SupportingTokens.request:krb_token.CustomToken.WssCustomToken.localname
 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ] ]'
```

3. Specify the authentication token for the policy type.

   Use the setPolicyType command to specify the Uniform Resource Identifier (URI) of the authentication token for services as the value for the `SupportingTokens.request:krb_token.CustomToken.WssCustomToken.uri` property. Use the `[ ]` syntax to specify an empty string. The following example specifies an empty string as the value for the authentication token:

```
AdminTask.setPolicyType('-policySet AuthenticationTokenService -policyType
 WSSecurity -attributes "[ [SupportingTokens.request:krb_token.CustomToken.IncludeToken
 http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient] [enabled true] [type
 WSSecurity] [description  [Policies for sending security tokens and providing message confidentiality and integrity,
 based on the OASIS Web Service Security and Token Profiles specifications.]]
 [SupportingTokens.request:krb_token.CustomToken.WssCustomToken.uri []] [provides []]
 [SupportingTokens.request:krb_token.CustomToken.WssCustomToken.localname
 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ] ]"')
```

- Configure custom bindings for security tokens.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Display the properties of the bindings of interest.

     Use the getBinding command to display detailed property information for the binding of interest, as the following command demonstrates:

```
AdminTask.getBinding('-policyType WSSecurity -bindingLocation "" -bindingName
 AuthenticationTokenService')
```

     The getBinding command returns a properties object that contains name and value pairs for each property, as the following sample output displays:

```
'[ [application.securityinboundbindingconfig.tokenconsumer_0.properties_0.name
 com.ibm.wsspi.wssecurity.krbtoken.serviceSPN]  [application.securityinboundbindingconfig.tokenconsumer_0.valuetype.localname
 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ]
 [application.securityinboundbindingconfig.tokenconsumer_0.valuetype.uri ]
 [application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.classname
 com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler]  [application.name
 application][application.securityinboundbindingconfig.tokenconsumer_0.properties_0.value HTTP/derekho1.firehorse.austin.ibm.com]
 [application.securityinboundbindingconfig.tokenconsumer_0.jaasconfig.configname system.wss.consume.KRB5BST]
 [application.securityinboundbindingconfig.tokenconsumer_0.name
 con_krbtoken][application.securityinboundbindingconfig.tokenconsumer_0.classname
 com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenConsumer]
 [application.securityinboundbindingconfig.tokenconsumer_0.securitytokenreference.reference request:krb_token] ]'
```

  3. Specify the authentication token for the policy type.

     Use the setBinding command to specify the Uniform Resource Identifier (URI) of the authentication token for services as the value for the `application.securityinboundbindingconfig.tokenconsumer_0.valuetype.uri` property. Use the `[ ]` syntax to specify an empty string. The following example specifies an empty string as the value for the authentication token:

```
AdminTask.setBinding('-policyType WSSecurity -bindingLocation ""
 -bindingName AuthenticationTokenService -attributes  "[
 [application.securityinboundbindingconfig.tokenconsumer_0.properties_0.name com.ibm.wsspi.wssecurity.krbtoken.serviceSPN]
 [application.securityinboundbindingconfig.tokenconsumer_0.valuetype.localname
 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ]
 [application.securityinboundbindingconfig.tokenconsumer_0.valuetype.uri []]
 [application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.classname
 com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler] [application.name
 application][application.securityinboundbindingconfig.tokenconsumer_0.properties_0.value
 HTTP/derekho1.firehorse.austin.ibm.com] [application.securityinboundbindingconfig.tokenconsumer_0.jaasconfig.configname
 system.wss.consume.KRB5BST][application.securityinboundbindingconfig.tokenconsumer_0.name
 con_krbtoken][application.securityinboundbindingconfig.tokenconsumer_0.classname
 com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenConsumer]
 [application.securityinboundbindingconfig.tokenconsumer_0.securitytokenreference.reference request:krb_token]
 ]"')
```

Results

If the setPolicyType and setBinding commands return a 'true' value, the system successfully updated the policy and binding configurations.

Related concepts

WSSecurity default policy sets
The WSSecurity default policy sets are based on the Web Services Security (WS-Security) 1.0 and Web Services Addressing (WS-Addressing) specifications. The WSSecurity default policy sets include the WSSecurity default policy set, the Lightweight Third-Party Authentication (LTPA) WSSecurity policy set, and the Username WSSecurity policy set. Use the WSSecurity default policy sets to build secure Web services.

Related tasks

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

"WSSecurity policy and binding properties" on page 1083
Use the **attributes** parameter for the setPolicyType and setBinding commands to specify additional configuration information for the WSSecurity policy and binding configurations. Application and system policy sets can use the WSSecurity policy and binding configuration.

Creating policy set attachments using the wsadmin tool

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to create policy set attachments. If you have access to a specific resource only, you can create policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to create policy set attachments. If you have access to a specific resource only, you can create policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can create policy set attachments for application resources only.
Operator	The Operator role cannot create policy set attachments.
Monitor	The Monitor role cannot create policy set attachments.

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

About this task

To use a new policy set to manage policies for your application, you must attach the policy set to an application artifact or artifacts. When the application restarts, the application uses the policies from the newly attached policy set.

1. Launch a scripting command.
2. Select an application with Web services to update. Use the listWebServices command to list all Web services and the associated applications. Enter the following command to list all Web services and attributes:

```
AdminTask.listWebServices()
```

   For each Web service, the command returns the associated application name, module name, service name, and service type. For example, the following information is returned:

```
'[ [service {http://www.ibm.com}service1] [client false] [application  application1]
 [module webapp1.war] [type JAX-WS] ]'
```

3. Create a policy set attachment for an application.

   For the commands in the PolicySetManagement group, the term **resource** refers to a Web service artifact. For application and service client policy sets, the artifacts use the application hierarchy. The application hierarchy includes a Web service, module name, endpoint, or operation. Enter the value for the -resource parameter as a string, with a backslash (/) character as a delimiter.

   Note: When attempting to connect to a Web service from a thin client, verify that the resources you are specifying are valid before running the updatePolicySetAttachment command. No configuration changes are made if the requested resource does not match a resource in the attachment file for the application.

   Use the following format for application and client policy set attachments:

   - `WebService:/`

     Attaches all artifacts in the application to the policy set.
   - `WebService:/webapp1.war:{http://www.ibm.com}myService`

     Attaches all artifacts within the Web service `{http://www.ibm.com}myService` to the policy set. You must provide a fully qualified name (QName) for the service.
   - `WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA`

     Attaches all operations for the `endpointA` endpoint to the policy set.
   - `WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA/operation1`

     Attaches only the `operation1` operation to the policy set.

   The format for the -resource string differs for system policy set attachments for the trust service. Use the following format for system policy set attachments:

   - `Trust.opName:/`

     The `opName` attribute can be issue, renew, cancel, or validate.
   - `Trust.opName:/url`

     The `opName` attribute can be issue, renew, cancel, or validate. You can specify any valid URL for the `url` attribute.

   a. Enter the command to attach the policy set to the application. This command attaches the `policyset1` application policy set to all artifacts in the `WebService` application.

      Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy

set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

To attach a policy set to a Web service application, specify the `provider` value for the -attachmentType parameter:

```
AdminTask.createPolicySetAttachment('[-policySet policyset1 -resources
 "WebService:/" -applicationName WebService -attachmentType provider]')
```

To attach a policy set to a service client application, specify the `client` value for the -attachmentType parameter, as the following example demonstrates:

```
AdminTask.createPolicySetAttachment('[-policySet policyset1 -resources
 "WebService:/" -applicationName WebService -attachmentType client]')
```

To create a trust service attachment for a system policy set, specify the `provider` value for the -attachmentType parameter and the `[systemType trustService]` value for the -attachmentProperties parameter, as the following example demonstrates:

```
AdminTask.createPolicySetAttachment('[-policySet policyset1 -resources
 "WebService:/" -applicationName WebService -attachmentType client -attachmentProperties "[systemType
 trustService]"]')
```

This command returns an attachment ID number that you must use to reference this attachment. In the next step, use the attachment ID number to set the binding configuration. For this example, the attachment ID number is `124`.

4. Run the command to set the binding. The following example demonstrates how to set the timestamp expiration attribute on the `SecureConversation123binding` binding for the `WSSecurity` policy, on the `WebService` Web service application. To attach a policy set to a Web services application, specify the `provider` value for the -attachmentType parameter.

```
AdminTask.setBinding('-policyType WSSecurity -bindingLocation "[ [application WebService] [attachmentId 124] ]" -attachmentType provider
 -bindingName SecureConversation123binding -attributes "[application.securityoutboundbindingconfig.timestampexpires.expires 5]"')
```

5. Save the configuration changes.

   Enter the following command to save your changes:

```
AdminConfig.save()
```

Results

Your have attached the policy set to the application artifact or artifacts specified. Restart your application to use the policies from the newly attached policy set.

What to do next

Manage and update your attachments.

Related tasks

Configuring attachments for the trust service using the administrative console
You can attach the trust service operations for a service endpoint to a system policy set and binding. Each new endpoint that is specified initially has the following four operations: issue, renew, cancel, and validate. By default, all endpoints inherit the policy set and binding that are attached to the respective trust service operation under Trust Service Defaults. However, you can explicitly attach a different policy set.

"Managing policy set attachments using the wsadmin tool"
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

Search attached applications collection
Use this page to search for applications and other resources that are attached to a specific policy set or to search for applications and other resources that have attached service resources.

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

"WebServicesAdmin command group for the AdminTask object" on page 1018
Use this topic as a reference for the commands for the WebServicesAdmin group of the AdminTask object. Use these commands with your administrative scripts to list available Web services, list web services attributes, determine the endpoint configuration for a Web service, and determine a specific operation name.

Managing policy set attachments using the wsadmin tool

Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to manage policy set attachments. If you have access to a specific resource only, you can manage policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to manage policy set attachments. If you have access to a specific resource only, you can manage policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can manage policy set attachments for application resources only.
Operator	The Operator role cannot manage policy set attachments.
Monitor	The Monitor role cannot manage policy set attachments.

About this task

Policy set attachments define how a policy set is attached to resources and binding configurations.

- Query the configuration for policy set attachments and attachment properties.

  Before making configuration changes to your policy set attachments, use the listAttachmentsForPolicySet and getPolicySetAttachments commands to view current configuration information about your policy set attachments.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Use the listAttachmentsForPolicySet command to view all applications to which a specific policy set is attached, for example:

     ```
     AdminTask.listAttachmentsForPolicySet('[-policySet PolicySet1]')
     ```

     Use the -attachmentType parameter to narrow your query. You can query for `provider` or `client` attachments.

     Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.
  3. Use the getPolicySetAttachments command to view the properties for all policy set attachments in a specified application, for example:

     ```
     AdminTask.getPolicySetAttachments('[-applicationName application1]')
     ```

     Use the -attachmentType parameter to narrow your query. You can query for `provider` or `client` attachments.
- Determine the assets to which a specific policy set is attached.

  Use the listAssetsAttachedToPolicySet command to display the assets that are attached to the policy set of interest, as the following example demonstrates:

  ```
  AdminTask.listAssetsAttachedToPolicySet('[-policySet SecureConversation]')
  ```

  The command returns a list of properties that describe each asset. Each properties object contains the assetType property, which specifies the type of asset.
- Modify resources that apply to a policy set attachment.
  1. Launch the wsadmin scripting tool using the Jython scripting language.

2. Determine the resource of interest and review the command syntax for the updatePolicySetAttachment command.

   For the commands in the PolicySetManagement group, the term **resource** refers to a Web service artifact. For application and service client policy sets, the artifacts use the application hierarchy. The application hierarchy includes a Web service, module name, endpoint, or operation. Enter the value for the -resource parameter as a string, with a backslash (/) character as a delimiter.

   Note: When attempting to connect to a Web service from a thin client, verify that the resources you are specifying are valid before running the updatePolicySetAttachment command. No configuration changes are made if the requested resource does not match a resource in the attachment file for the application.

   Use the following format for application and client policy set attachments:

   – `WebService:/`

     Attaches all artifacts in the application to the policy set.

   – `WebService:/webapp1.war:{http://www.ibm.com}myService`

     Attaches all artifacts within the Web service `{http://www.ibm.com}myService` to the policy set. You must provide a fully qualified name (QName) for the service.

   – `WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA`

     Attaches all operations for the `endpointA` endpoint to the policy set.

   – `WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA/operation1`

     Attaches only the `operation1` operation to the policy set.

   The format for the -resource string differs for system policy set attachments for the trust service. Use the following format for system policy set attachments:

   – `Trust.opName:/`

     The `opName` attribute can be issue, renew, cancel, or validate.

   – `Trust.opName:/url`

     The `opName` attribute can be issue, renew, cancel, or validate. You can specify any valid URL for the `url` attribute.

3. Modify the attachment.

   For example, the policy set attachment is connected to the operation1 operation, which is a specific single operation. To attach the 124 attachment to all operations for the endpointA endpoint, enter the following command:

   ```
   AdminTask.updatePolicySetAttachment('[-attachmentId 124 -resources
   "WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA" -applicationName application1]')
   ```

   Note: The updatePolicySetAttachment command replaces all existing resources for an attachment with the resources specified in the command. You can also update your policy set attachments using the addToPolicySetAttachment command to add resources to an existing attachment or the createPolicySetAttachment command to create an attachment for a specific resource. For more information about these commands reference the commands for the PolicySetManagement group for the AdminTask object.

4. Save the configuration changes.

   Enter the following command to save your changes:

   ```
   AdminConfig.save()
   ```

- Remove resources that apply to a policy set attachment.
  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine which resources to remove with the command. You can remove a resource for each Web service artifact, each operation for an endpoint, or for a specific operation. In the following example, the command removes the `newAttach` attachment from `operation1`, which is associated with the `plantShop` application.

```
AdminTask.removeFromPolicySetAttachment('[-attachmentId newAttach -resources
"WebService:/webapp1.war:{http://www.ibm.com}myPlantService/endpointA/operation1" -applicationName plantShop]')
```

The command returns a success or failure message.

3. Save the configuration changes.

   Enter the following command to save your changes:

   ```
   AdminConfig.save()
   ```

- Transfer attachments from one policy set to another policy set. This command detaches each Web service from the source policy set and attaches those Web services to the destination policy set. The destination policy set must have the same set of enabled policy types as the source policy set.

  1. Enter the following command to transfer all attachments:

     ```
     AdminTask.transferAttachmentsForPolicySet('[-sourcePolicySet PolicySet1
     -destinationPolicySet PolicySet2]')
     ```

     The command returns a success or failure message.

  2. Save the configuration changes.

     Enter the following command to save your changes:

     ```
     AdminConfig.save()
     ```

Related tasks

Configuring attachments for the trust service using the administrative console
You can attach the trust service operations for a service endpoint to a system policy set and binding. Each new endpoint that is specified initially has the following four operations: issue, renew, cancel, and validate. By default, all endpoints inherit the policy set and binding that are attached to the respective trust service operation under Trust Service Defaults. However, you can explicitly attach a different policy set.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

Search attached applications collection
Use this page to search for applications and other resources that are attached to a specific policy set or to search for applications and other resources that have attached service resources.

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Configuring general, cell-wide bindings for policies using the wsadmin tool

You can use the Jython or Jacl scripting language to customize your cell-wide default binding configuration. Create multiple cell-wide general bindings that you can attach to applications.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure bindings. If you have access to a specific resource only, you can configure bindings for the resource for which you have access. Only the Administrator role can edit binding attributes.
Configurator	The Configurator role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Deployer	The Deployer role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Operator	The Operator role can view, but cannot configure bindings.
Monitor	The Monitor role can view, but cannot configure bindings.

.

About this task

Bindings are environment and platform-specific information such as key store information, keys used for signature and encryption, or authentication information.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Use the following guidelines to manage bindings in your environment:

- To display or modify default Version 6.1 bindings, Version 7.0 trust service bindings, or to reference bindings by attachment for an application, specify the attachmentId and bindingLocation parameters with the getBinding or setBinding commands.
- To use or modify general Version 7.0 bindings, specify the bindingName parameter with the getBinding or setBinding commands.
- To display the version of a specific binding, specify the **version** attribute for the getBinding command.

Use a Version 6.1 binding for an application in a Version 7.0 environment if:

- The module in the application is installed on at least one Web Services Feature Pack server.
- The application contains at least one Version 6.1 application-specific binding. The application server does not assign general bindings to resource attachments for applications that are installed on a Web Services Feature Pack server. All application-specific bindings for an application must be at the same level.

General service provider and client bindings are not linked to a particular policy set and they provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that

are deployed to a server to share binding configuration. You can also accomplish this sharing of binding configuration by assigning the binding to each application deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:

1. Server level default
2. Security domain level default
3. Global security (cell) default

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the policy to update.

   To view a list of all available policies for a specific policy set, use the listPolicyType command. For example:

```
AdminTask.listPolicyTypes('[-policySet PolicySet1]')
```

3. Retrieve the current binding configuration for the policy to determine the attributes to update.

   Use the getBinding command to display a Properties object containing all configuration attributes for a specific policy binding. Specify a Properties object for the -bindingLocation parameter using an empty Properties object. For example:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "" -bindingName cellWideBinding1')
```

   To return a specific configuration attribute for the policy, use the -attributes parameter. For example, enter this command to determine if the WSAddressing policy has workload management enabled:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "" -bindingName cellWideBinding1 -attributes "[preventWLM]"')
```

   The command returns a properties object which contains the value of the requested attribute, preventWLM.

4. Edit the binding configuration.

   Use the setBinding command to update your binding configuration for a policy. To specify that you are editing a cell-wide binding, set the -bindingLocation parameter by passing a null or empty Properties object and specify the name of the binding with the -bindingName parameter. You can further customize your binding with the following parameters:

Parameter	Description	Data type
-policyType	Specifies the policy of interest.	String, optional
-attributes	Specifies the attribute values to update. This parameter can include all binding attributes for the policy or a subset to update.	Properties, optional
-replace	Specifies whether to replace all of the existing binding attributes with the attributes specified in the command. Use this parameter to remove optional parts of the configuration for policies with complex data. The default value is false.	Boolean, optional

Parameter	Description	Data type
-remove	Use this parameter to remove a specific policy from the binding configuration. The default value for the remove parameter is false. If the policyType parameter is not specified, the command removes the custom binding from the attachment. To delete the binding configuration, provide a value for the bindingName parameter and an asterisk character (*) for the attachmentId.	Boolean, optional
-domainName	Specifies the domain name for the binding. Use this parameter to scope a binding to a domain other than the global security domain.	String, optional

You must use the -attributes parameter when editing your binding configuration for cell-wide bindings. The following example disables workload management within the cell-wide default binding for the `WSAddressing` policy:

```
AdminTask.setBinding('-policyType WSAddressing -bindingLocation "" -bindingName cellWideBinding1 -attributes "[preventWLM false]"')
```

5. Save your configuration changes.

```
AdminConfig.save()
```

Related tasks

Reassigning bindings to policy sets
After you create a custom attachment binding, you can reassign that binding to another service artifact if necessary. You can reset a service artifact, such as an application, service, or endpoint to use the inherited bindings or default bindings.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

Policy set bindings settings
Use this page to view or define general or application specific bindings configuration information that is specific to a system for policies that you can associate with the selected policy set. Use the links on this page to work with bindings for each specific policy.

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Configuring Version 6.1 server-specific default bindings for policies using the wsadmin tool

You can use the Jython or Jacl scripting language to customize WebSphere Application Server Version 6.1 server-specific default bindings for policies to match your installation environment or requirements.

Before you begin

Server-specific default bindings use the WebSphere Application Server Version 6.1 namespace.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure bindings. If you have access to a specific resource only, you can configure bindings for the resource for which you have access. Only the Administrator role can edit binding attributes.
Configurator	The Configurator role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.

Administrative role	Authorization
Deployer	The Deployer role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Operator	The Operator role can view, but cannot configure bindings.
Monitor	The Monitor role can view, but cannot configure bindings.

About this task

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Use the following guidelines to manage bindings in your environment:

- To display or modify default Version 6.1 bindings, Version 7.0 trust service bindings, or to reference bindings by attachment for an application, specify the attachmentId and bindingLocation parameters with the getBinding or setBinding commands.
- To use or modify general Version 7.0 bindings, specify the bindingName parameter with the getBinding or setBinding commands.
- To display the version of a specific binding, specify the **version** attribute for the getBinding command.

Use a Version 6.1 binding for an application in a Version 7.0 environment if:

- The module in the application is installed on at least one Web Services Feature Pack server.
- The application contains at least one Version 6.1 application-specific binding. The application server does not assign general bindings to resource attachments for applications that are installed on a Web Services Feature Pack server. All application-specific bindings for an application must be at the same level.

General service provider and client bindings are not linked to a particular policy set and they provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that are deployed to a server to share binding configuration. You can also accomplish this sharing of binding configuration by assigning the binding to each application deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:

1. Server level default
2. Security domain level default
3. Global security (cell) default

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the policy to update.

   To view a list of all available policies for a specific policy set, use the listPolicyTypes command, as the following example demonstrates:

```
AdminTask.listPolicyTypes('[-policySet WSAddressing]')
```

3. Retrieve the current binding configuration for the policy to determine the attributes to update.

   Use the getBinding command to display a Properties object containing all configuration attributes for a specific policy binding. Specify a Properties object for the -bindingLocation parameter using the property names `node` and `server`. For example:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[node node1]
[server server1]]"')
```

   To return a specific configuration attribute for the policy, use the -attributes parameter. For example, enter this command to determine if the policy is enabled:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[node node1]
[server server1]]" -attributes "[[preventWLM]]"')
```

   The command returns a properties object which contains the value of the requested attribute, `preventWLM`.

4. Edit the binding configuration.

   Use the setBinding command to update your binding configuration for a policy. To specify that you are editing a server-specific default binding, set the -bindingLocation parameter using the `node` and `server` property names in a Properties object. You can further customize your binding with the following optional parameters:

Parameter	Description	Data type
-policyType	Specifies the policy of interest.	String, optional
-remove	Use this parameter to remove a server-level binding configuration. The default value for the -remove parameter is `false`.	Boolean, optional
-attributes	Specifies the attribute values to update. This parameter can include all binding attributes for the policy or a subset to update. The -attributes parameter is not required if you are removing your server-level binding.	Properties, optional
-replace	Specifies whether to replace all of the existing binding attributes with the attributes specified in the command. Use this parameter to remove optional parts of the configuration for policies with complex data. The default value is `false`.	Boolean, optional
-domainName	Specifies the domain name for the binding. Use this parameter to scope a binding to a domain other than the global security domain.	String, optional

   You should always specify the -attributes parameter when editing your configuration. The following example disables workload management within the server-specific default binding for the `WSAddressing` policy:

```
AdminTask.setBinding('-policyType WSAddressing -bindingLocation "[ [server server1] [node node01] ]" -attributes "[preventWLM false]"')
```

5. Save your configuration changes.

```
AdminConfig.save()
```

Related tasks

Creating application specific bindings for policy set attachment
After you attach a policy set to a service artifact such as an application, service, or endpoint, you can define application specific bindings for the attached policy set.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Configuring application-specific and system bindings using the wsadmin tool

Use the Jython or Jacl scripting language to edit custom application bindings and system bindings for policies to match your installation environment or system requirements.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure bindings. If you have access to a specific resource only, you can configure bindings for the resource for which you have access. Only the Administrator role can edit binding attributes.
Configurator	The Configurator role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Deployer	The Deployer role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Operator	The Operator role can view, but cannot configure bindings.
Monitor	The Monitor role can view, but cannot configure bindings.

About this task

Binding configurations are environment- and platform-specific information such as keystore information, keys used for signature and encryption, or authentication information. You can use the default binding for each policy set or define application-specific bindings within an application.

There are three types of bindings to use with your policy sets, including cell-level, application server level, and application-level. Default bindings are used at the cell-level or application server level. This topic refers to system binding information or bindings that are defined at the application level, which overrides the cell-level or application server level definition.

Use default bindings only to develop and test applications. You must change signing and encryption keys before using your bindings in a production environment.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Use the following guidelines to manage bindings in your environment:

- To display or modify default Version 6.1 bindings, Version 7.0 trust service bindings, or to reference bindings by attachment for an application, specify the attachmentId and bindingLocation parameters with the getBinding or setBinding commands.
- To use or modify general Version 7.0 bindings, specify the bindingName parameter with the getBinding or setBinding commands.
- To display the version of a specific binding, specify the **version** attribute for the getBinding command.

Use a Version 6.1 binding for an application in a Version 7.0 environment if:

- The module in the application is installed on at least one Web Services Feature Pack server.

- The application contains at least one Version 6.1 application-specific binding. The application server does not assign general bindings to resource attachments for applications that are installed on a Web Services Feature Pack server. All application-specific bindings for an application must be at the same level.

General service provider and client bindings are not linked to a particular policy set and they provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that are deployed to a server to share binding configuration. You can also accomplish this sharing of binding configuration by assigning the binding to each application deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:

1. Server level default
2. Security domain level default
3. Global security (cell) default

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the current binding data for the attachment of interest.

   Use the getPolicySetAttachments command to determine the attachment ID. You will need to specify the attachment ID in the getBinding and setBinding commands to specify that this is a application-specific binding configuration. Use the following command to retrieve the attachment ID:

   ```
   AdminTask.getPolicySetAttachments('-applicationName application1')
   ```

   Use the getBinding command to display a properties object that contains each configuration attribute for a specific policy binding configuration. For application and client policy set attachments, specify a properties object for the -bindingLocation parameter using the `application` and `attachmentId` property names. For a system policy set attachment for the trust service, specify only the `attachmentId` property name. The following example queries for an application policy set binding configuration:

   ```
   AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[application application1][attachmentId 123]]"')
   ```

   To return a specific configuration attribute for the policy, use the -attributes parameter.
3. Edit the binding configuration.

   Use the setBinding command to update your binding configuration for a policy. To specify that you are editing a application-specific binding configuration, set the -bindingLocation parameter by specifying the `application` and `attachmentId` property names in a properties object. You can additionally specify the -attachmentType parameter as `provider` or `client`.

   Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

Customize your binding configuration with the following optional parameters:

Parameter	Description	Data type
-policyType	Specifies the policy of interest.	String, optional
-remove	Use this parameter to remove a specific policy from the binding configuration. The default value for the -remove parameter is false. If the -policyType parameter is not specified, the command removes the application-specific binding from the attachment. To delete the binding configuration, provide a value for the -bindingName parameter and an asterisk character (*) for the -attachmentId parameter.	Boolean, optional
-attributes	Specifies the attribute values to update. This parameter can include each binding configuration attribute for the policy or a subset of attributes to update. If you do not specify the attributes parameter, the command only updates the binding configuration location that the specified attachment uses.	Properties, optional
-bindingName	Specifies the name for the binding configuration. Use this parameter to specify a name for the binding when you create a new application-specific binding. You can also use this parameter to switch an attachment to use a different, existing application-specific binding configuration. Lastly, you must specify a value for this parameter to delete a binding configuration.	String, optional
-replace	Specifies whether to replace all of the existing binding configuration attributes with the attributes specified in the command. Use this parameter to remove optional parts of the configuration for policies with complex data. The default value is `false`.	Boolean, optional
-domainName	Specifies the domain name for the binding. Use this parameter to scope a binding to a domain other than the global security domain.	String, optional

The following example disables workload management for the `myApplication` application's binding configuration for the `WSAddressing` policy:

```
AdminTask.setBinding('[-policyType WSAddressing -bindingLocation "[ [application myApplication] [attachmentId 123] ]"
 -attributes "[preventWLM false]" -attachmentType provider]')
```

4. Save the configuration changes.

   Enter the following command to save your changes.

   ```
   AdminConfig.save()
   ```

Related tasks

Creating application specific bindings for policy set attachment
After you attach a policy set to a service artifact such as an application, service, or endpoint, you can define application specific bindings for the attached policy set.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Creating application-specific and trust service-specific bindings using the wsadmin tool

You can use the Jython or Jacl scripting language to create application-specific and trust service-specific bindings to match your installation environment or requirements.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure bindings. If you have access to a specific resource only, you can configure bindings for the resource for which you have access. Only the Administrator role can configure binding attributes.
Configurator	The Configurator role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Deployer	The Deployer role with cell-wide or resource specific access can assign or unassign bindings, but cannot edit attributes.
Operator	The Operator role can view, but cannot configure bindings.
Monitor	The Monitor role can view, but cannot configure bindings.

About this task

Policy set bindings specify the details about how your quality of service (QoS) is configured. For example, a policy set attachment determines that sign, encrypt, or reliable messaging should be enabled. The policy set binding specifies how the protection is configured, for example, the path of the keystore file, the class name of the token generator, or the Java™ Authentication and Authorization Service (JAAS) configuration name.

For application policy sets, you can specify the policy set bindings at the cell-level using default binding configurations, at the application level using application-specific binding configurations, or at the cell-level with general bindings. Server-level default bindings are deprecated. If no binding information is specified during policy set attachment, the policy set inherits the default binding. In WebSphere Application Server Version 7.0, you can specify a general binding as the default for a server instead of server-default bindings.

For system policy sets, you can specify the bindings at the cell-level and the server-level. The available bindings for system policy sets are the TrustServiceSymmetricDefault and TrustServiceSecurityDefault bindings. If no custom binding information is specified by the attachment, the resources inherit the TrustServiceSymmetricDefault or TrustServiceSecurityDefault binding.

Note: Only use default binding for development and testing. You must customize the signing and encryption keys in your binding configurations for a production environment.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Use the following guidelines to manage bindings in your environment:

- To display or modify default Version 6.1 bindings, Version 7.0 trust service bindings, or to reference bindings by attachment for an application, specify the attachmentId and bindingLocation parameters with the getBinding or setBinding commands.
- To use or modify general Version 7.0 bindings, specify the bindingName parameter with the getBinding or setBinding commands.
- To display the version of a specific binding, specify the **version** attribute for the getBinding command.

Use a Version 6.1 binding for an application in a Version 7.0 environment if:

- The module in the application is installed on at least one Web Services Feature Pack server.
- The application contains at least one Version 6.1 application-specific binding. The application server does not assign general bindings to resource attachments for applications that are installed on a Web Services Feature Pack server. All application-specific bindings for an application must be at the same level.

General service provider and client bindings are not linked to a particular policy set and they provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that are deployed to a server to share binding configuration. You can also accomplish this sharing of binding configuration by assigning the binding to each application deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:

1. Server level default
2. Security domain level default
3. Global security (cell) default

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Determine the type of binding to create.

   You can create application policy set bindings at the cell-level, server-level, or application-level, and trust service policy set bindings at the cell-level or server-level.
3. Retrieve the current binding configuration for the policy of interest.

   Use the getBinding command to display a Properties object containing all configuration attributes for a specific binding. Specify the location of the binding by passing a properties object using the -bindingLocation parameter and the following reference table:

Type of binding	**-bindingLocation parameter value**
Cell-level	`-bindingLocation ""`
Server-level (deprecated)	`-bindingLocation "[[node node1][server server1]]"`
Application	`-bindingLocation "[[application application1][attachmentId 123]]"`
Trust service	`-bindingLocation "[[systemType trustService] [attachmentId 123]]"`

Type of binding	-bindingLocation parameter value
WS-Notification client	-bindingLocation "[[bus myBus][WSNService myService][attachmentId 123]"

For this example, the command displays the current binding configuration for the WSAddressing policy, with the 123 attachment ID, for the application1 application:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[application application1][attachmentId 123]]"')
```

To return a specific configuration attribute for the policy, use the -attributes parameter. For example, enter this command to determine if workload management is enabled:

```
AdminTask.getBinding('-policyType WSaddressing -bindingLocation "[[application application1][attachmentId 123]]" -attributes "[preventWLM]"')
```

The command returns a properties object which contains the value of the requested attribute, preventWLM. You might receive an error message if the binding does not exist in your configuration.

4. Create a new application-specific binding for the policy of interest.

   Use the setBinding command to create a binding configuration for a policy. To specify that you are creating an application-specific binding, set the -bindingLocation parameter by passing the application and attachmentId property names in a properties object. If you are creating a system policy set binding for the trust service, you only need to specify the attachmentId property name. You can further customize your binding with the following parameters:

Parameter	Description	Data type
-policyType	Specifies the policy of interest.	String, optional
-attachmentType	Specifies the type of policy set attachment. If the attachment is for an application, you do not need to specify this parameter. **Note:** The application and system/trust values for the -attachmentType parameter are deprecated. Specify the provider value in place of the application value. For system policy set attachments, specify the provider value for the attachmentType parameter and the "[systemType trustService]" value for the -attachmentProperties parameter. For WSNClient attachments, specify the client value for the attachmentType parameter and the bus and WSNService properties with the -attachmentProperties parameter.	String, optional
-attributes	Specifies the attribute values to update. This parameter can include all binding attributes for the policy or a subset of attributes.	Properties, optional
-bindingName	Specifies the name for your new application-specific binding. A name is generated if it is not specified.	String, optional
-domainName	Specifies the domain name for the binding. Use this parameter to scope a binding to a domain other than the global security domain.	String, optional

The following example creates the WSAddressing1234binding attachment-specific binding for the WSAddressing policy, assigned to the application1 application attachment 123, and enables workload management:

```
AdminTask.setBinding('-policyType WSAddressing -bindingName
 WSAddressing123binding -bindingLocation "[ [application application1] [attachmentId 123] ]" -attributes
 "[preventWLM false]"')
```

5. Optional: Add application-specific binding properties.

   Use the setBinding command to add any additional custom properties for your application-specific binding. The application server provides custom properties that are specific to each quality of service. Use the following format to specify custom properties for the binding:

```
AdminTask.setBinding('[-bindingLocation "[ [application application1] [attachmentId 123]
 ]" -policyType WSAddressing -attributes "[[properties_x:name key_value] [properties_x:value
 value]"]')
```

6. Save your configuration changes.

```
AdminConfig.save()
```

Related tasks

Creating application specific bindings for policy set attachment
After you attach a policy set to a service artifact such as an application, service, or endpoint, you can define application specific bindings for the attached policy set.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Deleting application-specific bindings from your configuration using the wsadmin tool

You can use the Jython or Jacl scripting language to delete a custom application or system policy set binding from your configuration. You cannot delete cell-level default bindings.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to modify bindings. If you have access to a specific resource only, you can modify bindings for the resource for which you have access.
Configurator	The Configurator role cannot modify bindings.
Deployer	The Deployer role cannot modify bindings.
Operator	The Operator role cannot modify bindings.
Monitor	The Monitor role cannot modify bindings.

About this task

Policy set bindings specify the details about how your quality of service (QoS) is configured. For example, a policy set attachment determines that sign, encrypt, or reliable messaging is enabled. The policy set binding specifies how the protection is configured, for example, the path of the keystore file, the class name of the token generator, or the Java Authentication and Authorization Service (JAAS) configuration name.

For application policy sets, policy set bindings exist at the cell-level and server-level using default binding configurations, or at the application level using application-specific binding configurations. You can also specify cell-level general bindings. For system policy sets, bindings exist at the cell level and server level, or you can create application-specific bindings.

Use the following procedure to delete application-specific bindings for trust policy sets and application level bindings for application policy sets:

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Retrieve the current binding configuration for the policy of interest.

   Use the getBinding command to display a properties object that contains all configuration attributes for a specific binding. Specify the location of the binding by passing a properties object using the bindingLocation parameter and the following reference table:

Type of Binding	Value for the -bindingLocation parameter
Application	`-bindingLocation "[[application application1][attachmentId 123]]"`
Trust service	`-bindingLocation "[[attachmentId 123]]"`
WS-Notification client	`-bindingLocation "[[bus myBus][WSNService myService][attachmentId 123]"`
General binding	`-bindingLocation []`

   In this example, the command displays the current binding configuration for the `WSAddressing` policy, with the `123` attachmentId, for the `application1` application:

```
AdminTask.getBinding('[-policyType WSAddressing -bindingLocation  "[[application
 application1][attachmentId 123]]"]')
```

   To display general policy set bindings, identify the bindings by specifying the -bindingName parameter, as the following example demonstrates:

```
AdminTask.getBinding('[-bindingLocation [] -attachmentType application
 -bindingName "General Provider Binding"]')
```

3. Remove the binding of interest from each attachment.

   You cannot remove a binding from your configuration if that binding is referenced by one or more attachments. Modify and use the following example command to remove a binding from an attachment:

```
AdminTask.setBinding('[-bindingLocation "[[application application1][attachmentId
 123]]"  -remove true]')
```

4. Delete the binding of interest.

   Use the setBinding command to delete a application-specific binding configuration. Specify the binding of interest with the -bindingName parameter, an asterisk (*) for the -attachmentId property, and set the -remove parameter to `true`. The following example setBinding command removes the `WSAddressing123binding` application policy set binding:

```
AdminTask.setBinding('[-attachmentType application -bindingName
 WSAddressing123binding  -bindingLocation "[[application application1][attachmentId *]]" -remove
 true]')
```

   The following example setBinding command removes the `customTrust` trust service binding:

```
AdminTask.setBinding('[-attachmentType "system/trust" -bindingName
 customTrust  -bindingLocation "[attachmentId *]" -remove true]')
```

   The following example setBinding command removes the `General Provider Binding` general binding:

```
AdminTask.setBinding('[-attachmentType application -bindingName "General Provider
 Binding" -bindingLocation [] -bindingScope domain -remove true]')
```

Note: You cannot delete general bindings if an attachment references the binding, or if the binding is set as the default for a server or domain.

5. Save your configuration changes.

Results

The application-specific binding of interest is removed from your configuration.

Related tasks

“Removing policy set bindings using the wsadmin tool” on page 1074
You can use the Jython or Jacl scripting language to remove binding configurations for policies and resources to match your installation environment or requirements.

“Creating policy sets using the wsadmin tool” on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

“Adding and removing policies using the wsadmin tool” on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

“Creating policy set attachments using the wsadmin tool” on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

“Managing policy set attachments using the wsadmin tool” on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

“Removing policy set attachments using the wsadmin tool” on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

“PolicySetManagement command group for the AdminTask object” on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Importing and exporting policy sets to client or server environments using scripting

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to export and import application or system policy sets for Web services. The exportPolicySet command creates an archive file based on the policy set configuration, and the importPolicySet command imports a default policy set or policy set from an archive file.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to import and export policy sets.
Configurator	The Configurator role cannot import and export policy sets.
Deployer	The Deployer role cannot import and export policy sets.
Operator	The Operator role cannot import and export policy sets.
Monitor	The Monitor role cannot import and export policy sets.

About this task

You can use the exportPolicySet and importPolicySet commands to exchange system or application policy sets between servers or between a client and a provider. To reuse a policy set on a new server or client, export the policy set to an archive file, then import the archive file on the destination server or client. This topic provides examples for exporting a policy set, importing a policy set from an archive file, and importing a default policy set.

- Export an application or system policy set to an archive file.

  Use the exportPolicySet command to create an archive file for the policy set of interest. For example, the following command creates the customSC.zip archive file in the C:\IBM\WebSphere\AppServer\PolicySets\ directory for the customSecureConversation policy set:

  ```
  AdminTask.exportPolicySet('[-policySet customSecureConversation
  -pathName C:\IBM\WebSphere\AppServer\PolicySets\customSC.zip]')
  ```

- Move the policy set archive file to the destination environment.

  If you are exporting the policy set to a client environment, then place the archive file on the classpath of the client.

- Import a policy set from an archive file or import a default policy set.

  Use the importPolicySet command to import the archive file containing the policy set configuration of interest to the destination environment. You cannot import a policy set onto a server or client environment if the policy set already exists in the destination environment.

  For example, the following command creates a customSecureConversation policy set from the customSC.zip archive file:

  ```
  AdminTask.importPolicySet('[-importFile C:\IBM\WebSphere\AppServer\bin\customSC.zip]')
  ```

  Additionally, you can also use the importPolicySet command to import a default policy set onto a server environment, as the following example demonstrates:

  ```
  AdminTask.importPolicySet('[-defaultPolicySet SecureConversation -policySet copyOfdefaultSC]')
  ```

- Save the configuration changes.

  Enter the following command to save your changes:

  ```
  AdminConfig.save()
  ```

Removing policy set bindings using the wsadmin tool

You can use the Jython or Jacl scripting language to remove binding configurations for policies and resources to match your installation environment or requirements.

Before you begin

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to delete bindings. If you have access to a specific resource only, you can delete bindings for the resource for which you have access.
Configurator	The Configurator role can unassign bindings, but cannot delete bindings.
Deployer	The Deployer role can unassign bindings, but cannot delete bindings.
Operator	The Operator role cannot modify bindings.
Monitor	The Monitor role cannot modify bindings.

About this task

Use the following steps to remove specific policies from your application-specific binding configuration, or to remove your entire binding configuration. For both of these removal options, you must use the -bindingLocation parameter to specify whether you are deleting an application-specific binding, server-specific default binding, or a binding for the trust service. Use the following table for examples using Jython syntax when specifying the type of binding to modify or remove:

Type of binding	Value for the -bindingLocation parameter
Server-level (for Version 6.1 bindings only)	`-bindingLocation "[[node node1][server server1]]"`
Application	`-bindingLocation "[[application application1][attachmentId 123]]"`
Trust service binding	`-bindingLocation "[[systemType trustService] [attachmentId 123]]"`
WS-Notification client	`-bindingLocation "[[bus myBus][WSNService myService][attachmentId 123]"`
General bindings	`-bindingLocation []`

- Remove a policy from your application-specific binding configuration.

  Use the following steps to remove a specific policy from your binding configuration. If you remove the last policy remaining in your binding configuration, the command removes binding information from all attachments and deletes it from your configuration.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Review the binding configuration to edit.

     Use the getBinding command to view the attributes for the binding, as the following example demonstrates:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[application application1][attachmentId 1234]]"')
```

     If the binding of interest is not referenced by an attachment ID, specify an asterisk character (*) for the attachmentId parameter to view the attributes for the binding, as the following example demonstrates:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[application application1][attachmentId *]]"')
```

  3. Remove the policy from the binding configuration.

     Use the setBinding command with the -policyType and -remove parameters to remove the policy of interest from the binding configuration. For example, use the following command to remove the `WSAddressing` policy from the binding configuration for the `application1` application:

```
AdminTask.setBinding('-policyType WSAddressing -remove true -bindingLocation "[[application application1][attachmentId 1234]]"
```

     If the binding to delete is not referenced by an attachment ID, specify an asterisk character (*) for the attachmentId parameter to delete the binding, as the following example demonstrates:

```
AdminTask.setBinding('-policyType WSAddressing -remove true -bindingLocation "[[application application1][attachmentId *]]"')
```

4. Save your configuration changes.

- Remove binding configurations from an attachment.

  Use the following steps to remove a server-specific default binding or a custom binding. You cannot remove cell-level default bindings from your configuration. When a binding is removed from an attachment, the resource it was removed from will inherit the server-level default binding, if one is present, or the cell-level default binding if the server-level binding is not present. Use the following steps to remove a binding configuration:

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Verify the current binding configuration to delete.

     Before removing the binding from the attachment, use the getBinding command to view the attributes for the binding, as the following example demonstrates:

```
AdminTask.getBinding('-policyType WSAddressing -bindingLocation "[[application application1][attachmentId 123]]"')
```

  3. Remove the current binding configuration from the attachment.

     For this example, this command removes the bindings from the `123` attachment for the `application1` application:

```
AdminTask.setBinding('-bindingLocation "[[application application1][attachmentId 123]]" -remove true')
```

     If the binding to delete is not referenced by an attachment ID, specify an asterisk character (*) for the -attachmentId parameter to remove the binding, as the following example demonstrates:

```
AdminTask.setBinding('-bindingLocation "[[application application1][attachmentId *]]" -remove true')
```

     To remove a server-specific default binding, specify the node name and server name with the -bindingLocation parameter. Server specific default bindings are deprecated. For example, this command removes the server-level default binding for the WS-Addressing policy from the `server1` server on the `node1` node:

```
AdminTask.setBinding('-policyType WSAddressing -bindingLocation "[[node node1][server server1]]" -remove true')
```

  4. Save your configuration changes.

- Remove a policy from a general binding.

  Use the following steps to remove a

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Verify the current binding configuration to delete.

     Before removing the binding from the attachment, use the getBinding command to view the attributes for the binding, as the following example demonstrates:

```
AdminTask.getBinding('-policyType WSAddressing -bindingName "General Provider Binding" -bindingLocation []')
```

  3. Remove the general binding.

     For this example, this command removes the `General Provider Binding` general binding:

```
AdminTask.setBinding('-bindingLocation [] -bindingName "General Provider Binding" -remove true')
```

  4. Save your configuration changes.

Related tasks

Reassigning bindings to policy sets
After you create a custom attachment binding, you can reassign that binding to another service artifact if necessary. You can reset a service artifact, such as an application, service, or endpoint to use the inherited bindings or default bindings.

"Deleting application-specific bindings from your configuration using the wsadmin tool" on page 1071
You can use the Jython or Jacl scripting language to delete a custom application or system policy set binding from your configuration. You cannot delete cell-level default bindings.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool"
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Removing policy set attachments using the wsadmin tool

You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Before you begin

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to remove policy set attachments. If you have access to a specific resource only, you can remove policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to remove policy set attachments. If you have access to a specific resource only, you can remove policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can remove policy set attachments for application resources only.
Operator	The Operator role cannot remove policy set attachments.
Monitor	The Monitor role cannot remove policy set attachments.

Determine which applications and policy sets to remove, detach, or transfer. Use the listWebServices command to list all Web services and for the application to edit. Enter the command to list all Web services and attributes for a specific application.

```
AdminTask.listWebServices('[–application application_name]')
```

To view a list of all Web services and associated applications, do not provide the -application parameter. For each Web service, the command returns the associated application name, module name, service name, and service type. You can also use the listAttachmentsForPolicySet and getPolicySetAttachments administrative commands to view existing configuration data. For additional information about these commands, use the information center topic for the PolicySetManagement group of commands for the AdminTask object.

About this task

There are four ways to remove policy set attachments, including:

- Remove a policy set attachment from an application.
- Remove resources that apply to a policy set attachment.
- Remove all attachments for a specific policy set and application.
- Transfer attachments between policy sets for a specific application.

Choose the appropriate procedure to remove your policy set attachments.

- Remove a policy set attachment from the application.
  1. Enter the following command to remove the policy set attachment from the application:

     ```
     AdminTask.deletePolicySetAttachment('[-attachmentId attachment_name -applicationName application_name]')
     ```

     The command returns a success or failure message. If you receive a success message, the policy set attachment is successfully removed from your application. If you receive a failure message, verify that the chosen policy set attachment exists in your configuration.
  2. Save the configuration changes.

     Enter the following command to save your changes:

     ```
     AdminConfig.save()
     ```
- Remove resources that apply to a policy set attachment.
  1. You can customize the command to remove resources for the Web service, endpoint, or operation.

     For the commands in the PolicySetManagement group, the term *resource* refers to a Web service artifact. For application and service client policy sets, the artifacts use the application hierarchy: Web service, module name, endpoint, or operation. Enter the value for the -resource parameter as a string, with a backslash (/) character as a delimiter. Use the following format for application and client policy set attachments:

WebService:/
Refers to all artifacts in the application to the policy set.

WebService:/webapp1.war:{http://www.ibm.com}myService
Refers to all artifacts within the Web service *{http://www.ibm.com}myService* to the policy set. You must provide a fully qualified name (QName) for the service.

WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA
Refers to all operations for the *endpointA* endpoint to the policy set.

WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA/operation1
Refers to only the *operation1* operation to the policy set.

The format for the **resource** string differs for system policy set attachments for the trust service. Use the following format for system policy set attachments:

Trust.<opName>:/
The *<opName>* attribute can be issue, renew, cancel, or validate.

Trust.<opName>:/url
The *<opName>* attribute can be issue, renew, cancel, or validate. You can specify any valid URL for the *url* attribute.

In the following example, the command removes the `attachment_name` attachment from the `operation1` operation, which is associated with the application, `application1`.

```
AdminTask.removeFromPolicySetAttachment('[-attachmentId attachment_name -resources
"WebService:/webapp1.war:{http://www.ibm.com}myService/endpointA/operation1" -applicationName application1]')
```

The command returns a success or failure message. You can also use the **updatePolicySetAttachments** command to remove attached resources.

2. Save the configuration changes.

   Enter the following command to save your changes:

   ```
   AdminConfig.save()
   ```

- Remove all attachments for a specific policy set.
  1. Remove application attachments for the policy set. To remove an attachment from an application, use the following command:

     ```
     AdminTask.deleteAttachmentsForPolicySet('[-policySet PolicySet1 -applicationName application1]')
     ```

     To remove all attachments from the policy set, use the following command:

     ```
     AdminTask.deleteAttachmentsForPolicySet('[-policySet PolicySet1]')
     ```

     Both commands return a success or failure message.
  2. Save the configuration changes.

     Enter the following command to save your changes:

     ```
     AdminConfig.save()
     ```

- Transfer attachments from one policy set to another policy set. This command detaches all Web services from the source policy set and attaches the Web services to the destination policy set.
  1. Enter the transferAttachmentsForPolicySet command to transfer all attachments within an application. Use the following command to transfer the attachments from the *PolicySet1* policy set to the *PolicySet2* policy set within the *application1* application:

     ```
     AdminTask.transferAttachmentsForPolicySet('[-sourcePolicySet PolicySet1
     -destinationPolicySet PolicySet2 -applicationName application1]')
     ```

     The command returns a success or failure message.
  2. Save the configuration changes.

     Enter this command to save your changes:

     ```
     AdminConfig.save()
     ```

Related tasks

Configuring attachments for the trust service using the administrative console
You can attach the trust service operations for a service endpoint to a system policy set and binding. Each new endpoint that is specified initially has the following four operations: issue, renew, cancel, and validate. By default, all endpoints inherit the policy set and binding that are attached to the respective trust service operation under Trust Service Defaults. However, you can explicitly attach a different policy set.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

"Creating policy set attachments using the wsadmin tool" on page 1049
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to define the policy set configuration for your Web services applications. You can attach policy sets to an application, Web service, endpoint, or specific operation.

"Managing policy set attachments using the wsadmin tool" on page 1052
Use the wsadmin tool to manage your policy set attachment configurations. You can use the Jython or Jacl scripting language to list all attachments and attachment properties, add or remove resources for an existing attachment, and transfer attachments across policy sets.

"Removing policy set attachments using the wsadmin tool" on page 1077
You can use the Jython or Jacl scripting language to remove and transfer policy sets from application artifacts. You can also remove resources that apply to a policy set attachment without deleting the policy set attachment.

Managing policy sets using the administrative console
You can use policy sets, or assertions that define services, to simplify your Web services configuration because policy sets group security and other Web services settings into reusable units. You can use the administrative console to create, modify, and delete custom policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Deleting policy sets using the wsadmin tool

Use the Jython or Jacl scripting language to delete policy sets from your configuration with the wsadmin tool. You must remove all policy set attachments before removing the policy set.

Before you begin

To complete this task, you must use the Administrator role with cell-wide access when administrative security is enabled.

Before deleting a policy set, you must delete or transfer all attachments to applications. You cannot delete default policy sets.

About this task

Use the following steps to delete custom policy sets from your configuration with the wsadmin tool:

1. Launch a scripting command.
2. List all policy sets in your configuration.
   - Enter the following command to list all application policy sets:
     ```
     AdminTask.listPolicySets()
     ```
   - Enter the following command to list all policy sets for the trust service:
     ```
     AdminTask.listPolicySets('[-policySetType system/trust]')
     ```
   - Enter the following command to list all system policy sets:
     ```
     AdminTask.listPolicySets('[-policySetType system]')
     ```
3. Determine which policy set to delete. Enter the following command to view the description and default indicator for a specific policy set:
   ```
   AdminTask.getPolicySet('[-policySet policySet_name]')
   ```
4. Delete the policy set.

   Enter the following command to delete a specific policy set.
   ```
   AdminTask.deletePolicySet('[-policySet PolicySet1)')
   ```

   The command returns a success or failure response. If you receive an error message, make sure that you have deleted all policy set attachments before entering the deletePolicySet command.
5. Save the configuration changes.

   Enter the command to save your changes.
   ```
   AdminConfig.save()
   ```

What to do next

If you deleted a policy set that was previously attached to an application, restart the affected application to update the configuration changes.

Related tasks

Deleting policy sets using the administrative console
You can use the administrative console to delete the default policy sets or the application specific policy sets that you have created.

"Configuring application and system policy sets for Web services using scripting" on page 1031
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure application or system policy sets for Web services. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

"Creating policy sets using the wsadmin tool" on page 1033
Create policy sets to centrally manage policies that are customized for your Web services. Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to create new policy sets, copy existing policy sets, or import a policy set configuration. You can also query for an existing policy set and respective attributes.

"Adding and removing policies using the wsadmin tool" on page 1037
You can use the Jython or Jacl scripting language and the wsadmin tool to query, add, and remove policies for your policy sets.

Related reference

"PolicySetManagement command group for the AdminTask object" on page 1107
You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Refreshing policy set configurations using scripting

Use the wsadmin tool to refresh the policy set configuration data. After refreshing the policy set configuration, the changes apply after restarting the application.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Get the object name of each PolicySetManager object.

   Use the completeObjectName option for the AdminControl object to set the object name for each PolicySetManager type object to the `objNameString` variable, as the following example demonstrates:

```
objNameString = AdminControl.completeObjectName('type=PolicySetManager,*')
```

3. Connect to the Managed Bean (MBean).

   The MBean supplies a remote interface to the MBean server that runs in the application server. The following example shows how to look up the MBean:

```
import javax.management as mgmt
```

4. Set the PolicySetManager MBean object name.

   The following example sets the PolicySetManager MBean object name to the `mbeanObj` variable, parameters to the `param` variable, and signature settings to the `sig` variable:

```
mbeanObj = mgmt.ObjectName(objNameString)
param=[]
sig=[]
```

5. Refresh the PolicySetManager MBean.

   The following example refreshes the policy set configuration:

```
AdminControl.invoke_jmx(mbeanObj, 'refresh', param, sig)
```

Example

The following example provides the Jython script that refreshes the policy set configuration:

```
objNameString = AdminControl.completeObjectName('type=PolicySetManager,*')
import javax.management as mgmt
mbeanObj = mgmt.ObjectName(objNameString)
param=[]
sig=[]
AdminControl.invoke_jmx(mbeanObj, 'refresh', param, sig)
```

Policy configuration properties for all policies

You can use the **attributes** parameter with the setPolicyType and setBinding commands to specify various properties for each quality of service (QoS) within a policy set. You can use the properties in this topic with each QoS within application and system policy sets.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

Attributes to configure for all QoS policies

Use the following list of attributes to configure attributes across all QoS policies using the Jython scripting language and the wsadmin tool:

enabled

Specifies whether the policy type is enabled or disabled. The following example provides the format to enter the attributes parameter:

```
-attributes "[[enabled true]]"
```

provides

Provides a description for your configuration. The following example provides the format to enter the attributes parameter:

```
-attributes "[[provides [Messaging Security]]]"
```

The following example uses the setPolicyType command to set the `enabled` and `provides` properties for the myCustomSecurityPS custom policy set, which contains a ReliableMessaging policy:

```
AdminTask.setPolicyType('[-policySet myCustomSecurityPS -policyType
WSReliableMessaging -attributes [[enabled true][provides
[Messaging security]]]]')
```

WSSecurity policy and binding properties

Use the **attributes** parameter for the setPolicyType and setBinding commands to specify additional configuration information for the WSSecurity policy and binding configurations. Application and system policy sets can use the WSSecurity policy and binding configuration.

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (″″). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

If the **attributes** parameter is not specified for the getPolicyType or getBinding command, the command returns all properties. If a partial property name is passed to the getPolicyType or getBinding command, the command returns all properties with names that start with the partial property name. For example, If SignatureProtection is passed to the getPolicyType command, the command returns all properties with names that start with ″SignatureProtection″, which might include:

```
SignatureProtection.response:
   int_body.SignedParts.Body,SignatureProtection.response:int_body.SignedParts.Header_0.Name
```

, and

```
SignatureProtection.response:int_body.SignedParts.Header_0.Namespace
```

.

There are an extensive number of combinations of settings that are available to secure your Web service applications. Because of the number of attributes and configuration options from the WS-Security Version 1.0 specification, all attributes are not defined in this topic. The following sections explain the hierarchy structure for the WSSecurity policy and binding attributes:

- WSSecurity policy properties
- WSSecurity binding properties
- “setPolicyType and setBinding command examples” on page 1090

WSSecurity policy properties

Use the getPolicyType command to review a properties object with the properties that are configured in your current WSSecurity policy file. Security policy schemata define the security assertions. Because the elements in the schema have hierarchical relationship, the property names for security policy also have the similar hierarchy. The hierarchical relationship between property names in the security policy is represented by a period (.) between two levels, concatenating the parent and child attributes. Examples of the properties include, but are not limited to, IncludeToken, Name, Namespace, XPath, XPathVersion. The following list describes the top-level assertion policy property names for the WSSecurity policy file:

AsymmetricBinding
You can specify zero or one binding assertion.

SymmetricBinding
You can specify zero or one binding assertion. AsymmetricBinding and SymmetricBinding cannot co-exist in a security policy file.

Wss11
You can specify zero or one Wss11 assertion.

Wss10
You can specify zero or one Wss10 assertion.

Trust10
You can specify zero or one Trust10 assertion.

SignatureProtection
You can specify zero or any number of signature protection assertions.

EncryptionProtection
You can specify zero or any number of encryption protection assertions

SupportingTokens
You can specify zero or any number of supporting token assertions.

For example, the following policy file example displays an AsymmetricBinding assertion:

```
<sp:AsymmetricBinding>
  <wsp:Policy>
    <sp:InitiatorSignatureToken>
      <wsp:Policy>
        <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy
  /200512/IncludeToken/AlwaysToRecipient">
          <wsp:Policy>
            <sp:WssX509V3Token10 />
          </wsp:Policy>
        </sp:X509Token>
      </wsp:Policy>
    </sp:InitiatorSignatureToken>
    <sp:RecipientSignatureToken>
      <wsp:Policy>
        <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy
  /200512/IncludeToken/AlwaysToInitiator">
          <wsp:Policy>
            <sp:WssX509V3Token10 />
          </wsp:Policy>
        </sp:X509Token>
      </wsp:Policy>
    </sp:RecipientSignatureToken>
    <sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:Basic256/>
      </wsp:Policy>
    </sp:AlgorithmSuite>
    <sp:Layout>
      <wsp:Policy>
        <sp:Strict/>
      </wsp:Policy>
```

```
      </sp:Layout>
    </wsp:Policy>
  </sp:AsymmetricBinding><sp:AsymmetricBinding>
    <wsp:Policy>
      <sp:InitiatorSignatureToken>
        <wsp:Policy>
          <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy
    /200512/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:WssX509V3Token10 />
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:InitiatorSignatureToken>
      <sp:RecipientSignatureToken>
        <wsp:Policy>
          <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy
    /200512/IncludeToken/AlwaysToInitiator">
            <wsp:Policy>
              <sp:WssX509V3Token10 />
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:RecipientSignatureToken>
    </wsp:Policy>
    <sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:Basic256/>
      </wsp:Policy>
    </sp:AlgorithmSuite>
    <sp:Layout>
      <wsp:Policy>
        <sp:Strict/>
      </wsp:Policy>
    </sp:Layout>
  </sp:AsymmetricBinding>
```

The AsymmetricBinding assertion returns the following property name and value pairs. The nested `wsp:Policy` layers are not displayed in the returned properties. Additionally, some properties return the `true` value which indicates that the WSSecurity configuration includes the related XML elements. To edit these properties, set the value as `true` to include the property, or set the value as an empty string,"", to remove the property.

```
AsymmetricBinding.Layout = Strict
AsymmetricBinding.AlgorithmSuite.Basic256 = true
AsymmetricBinding.RecipientSignatureToken.X509Token_0.IncludeToken = http://docs.oasis-open.org
/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToInitiator
AsymmetricBinding.InitiatorSignatureToken.X509Token_0.WssX509V3Token10 = true
AsymmetricBinding.InitiatorSignatureToken.X509Token_0.IncludeToken = http://docs.oasis-open.org
/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient
AsymmetricBinding.RecipientSignatureToken.X509Token_0.WssX509V3Token10 = true
```

Additionally, the following policy file example displays a SupportingTokens assertion:

```
<sp:SupportingTokens>
        <wsp:Policy wsu:Id="request:custom_auth">
            <spe:CustomToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
       ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient">
                <wsp:Policy>
                    <spe:WssCustomToken uri=http://bar.com/MyCustomToken localname="tokenv1">
                    </spe:WssCustomToken>
                </wsp:Policy>
            </spe:CustomToken>
        </wsp:Policy>
    </sp:SupportingTokens
```

The SupportingTokens assertion returns the following property name and value pairs. The nested `wsp:Policy` layers are not displayed in the returned property.

```
SupportingTokens.request:custom_auth.CustomToken_0.WssCustomToken.uri=http://bar.com
/MyCustomToken
SupportingTokens.request:custom_auth.CustomToken_0.IncludeToken=http://docs.oasis-open.org
/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient
SupportingTokens.request:custom_auth.CustomToken_0.WssCustomToken.localname=tokenv1
```

Note: The CustomToken property contains a subscript zero notation (_0) because the property might be displayed multiple times from the same type of token such as the RecipientSignatureToken or InitiatorSignatureToken tokens.

Although most property names follow the hierarchical relationship format described previously, the following exceptions exist:

- The wsu:Id element

  This element uses the actual value for the ID instead of using `Id` as the attribute name. The following policy file example property:

  ```
  <wsp:Policy wsu:Id="response:int_body">
    <sp:SignedParts>
      <sp:Body/>
    </sp:SignedParts>
  </wsp:Policy>
  ```

  The previous wsu:Id example returns the following properties:

  ```
  SignatureProtection.response:int_body.SignedParts.Body = true
  ```

- The Header element

  Because there can be multiple Header elements, the `Header_n` notation is used to represent this property. See the following policy file example:

  ```
  <wsp:Policy wsu:Id="request:conf_body">
     <sp:EncryptedParts>
       <sp:Body/>
       <sp:Header Name="MyElement" Namespace="http://foo.com/MyNamespace" />
     </sp:EncryptedParts>
   </wsp:Policy>
  ```

  The previous Header example returns the following properties:

  ```
  EncryptionProtection.request:conf_body.EncryptedParts.Header_0.Name=MyElement
  EncryptionProtection.request:conf_body.EncryptedParts.Header_0.Namespace=http://
  foo.com/MyNamespace
  ```

- The XPath element

  The `XPath_n` notation is used to represent this property because there can be multiple XPath elements. See the following policy file example:

  ```
  <wsp:Policy wsu:Id="request:int_body">
       <sp:SignedElements>
         <sp:XPath>SomeXPathExpression</sp:XPath>
         <sp:XPath>SomeOtherXPathExpression</sp:XPath>
       </sp:EncryptedElements>
     </wsp:Policy>
  ```

  The previous XPath example returns the following properties:

  ```
  SignatureProtection.request:int_body.SignedElements.XPath_0=SomeXPathExpression
  SignatureProtection.request:int_body.SignedElements.XPath_1=SomeOtherXPathExpression
  ```

- The X509Token element

  Use the X509Token_*n* notation to represent this property because multiple X509Token elements can exist. For an example, see the AsymmetricBinding assertion.

- The CustomToken element

  Use the CustomToken_*n* notation to represent this property because multiple CustomToken elements can exist. For an example, see the SupportingTokens assertion.

WSSecurity binding properties

Use the getBinding command to review a properties object with the properties that are configured in your current WSSecurity binding configuration. You can also use the administrative console to configure your WSSecurity bindings. Use the information center topics for configuring WSSecurity bindings with administrative console for more information.

The properties defined in this section reflect the hierarchy of the binding schema. Each part of the property name is a lowercase version of the schema type. For example, the application.securityinboundbindingconfig.tokenconsumer_0.jaasconfig.configname property follows the hierarchal format. The attributes begin with **application** or **bootstrap**. Attributes that begin with

application represent bindings that are associated with the main WS-Security policy. Attributes that begin with **bootstrap** represent bindings that are associated with the WS-Security bootstrap policy, where the WS-Security policy uses Secure Conversation.

Some property names might have an _n notation appended to them. This notation represents a list of items. For example, multiple **tokenconsumer** properties exist and are listed from tokenconsumer_*0* through tokenconsumer_*n*, where the set of **tokenconsumer** values are:

```
application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.
certpathsettings.certstoreref.reference
application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.
certpathsettings.trustanchorref.reference
application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.classname
application.securityinboundbindingconfig.tokenconsumer_0.classname
application.securityinboundbindingconfig.tokenconsumer_0.jaasconfig.configname
application.securityinboundbindingconfig.tokenconsumer_0.name
application.securityinboundbindingconfig.tokenconsumer_0.valuetype.localname
application.securityinboundbindingconfig.tokenconsumer_0.valuetype.uri
```

Additionally, some properties in the security binding file return a value of true when queried. To set these properties, set the value to true to include the property, or set the value to an empty string ("") to remove the property. For example, the time stamp, nonce, and trustAnyCertificate properties follow this pattern.

Use the setBinding command and the **attributes** parameter to add or remove properties to your WSSecurity binding configuration.

- To add a property, use the setBinding command to pass the property name with a non-zero length string value. To add a list item, use the _n notation to reflect a numeric value that is greater than any current numeric value for the property. For example, if the tokenconsumer_0 and tokenconsumer_1 properties exist in your configuration, specify the new tokenconsumer property as tokenconsumer_2. After adding a property, use the getBinding command to view the most recent list of configured properties.
- To remove a property, use the setBinding command to pass the property name with an empty string (""). For example, to remove all of the tokenconsumer_0 properties, specify the following property with the **attributes** parameter:

  ```
  application.securityinboundbindingconfig.tokenconsumer_0=""
  ```

  The previous example removes all properties that begin with the application.securityinboundbindingconfig.tokenconsumer_0 property name.

The following examples display several sets of properties to configure for your binding. This list does not include all properties to configure for the WSSecurity binding. Use this information as a reference to determine how to form specific property names.

signinginfo element

Use this property to configure signing information. For a custom binding, an unlimited number of **signinginfo** elements specified for the securityoutboundbindingconfig and securityinboundbindingconfig assertions can exist. In the default bindings, the system allows a maximum of two **signinginfo** elements for the securityoutboundbindingconfig and securityinboundbindingconfig assertions. The following example displays the format for two **signinginfo** elements:

```
application.securityinboundbindingconfig.signinginfo_0.signingkeyinfo_0
.reference=con_signkeyinfo
application.securityinboundbindingconfig.signinginfo_0.signingpartreference_0
.reference=request:int_body
application.securityoutboundbindingconfig.signinginfo_0.signingpartreference_0
.reference=response:int_body
application.securityoutboundbindingconfig.signinginfo_0.signingpartreference_0.timestamp=true
```

encryptioninfo element

Use this property to configure encryption information. For a custom binding, an unlimited number of **encryptioninfo** elements specified for the securityoutboundbindingconfig and securityinboundbindingconfig assertions can exist. In the default bindings, the system accepts a

maximum of two **encryptioninfo** elements for the `securityoutboundbindingconfig` and `securityinboundbindingconfig` assertions. The following example displays the format for two **encryptioninfo** properties:

```
application.securityinboundbindingconfig.encryptioninfo_0.encryptionpartreference
.nonce=true
application.securityinboundbindingconfig.encryptioninfo_0.encryptionpartreference
.reference=request:conf_body
application.securityoutboundbindingconfig.encryptioninfo_0.encryptionpartreference
.nonce=true
application.securityoutboundbindingconfig.encryptioninfo_0.encryptionpartreference
.timestamp=true
```

tokengenerator element

In the default bindings, the **tokengenerator** elements that the **signinginfo** or **encryptioninfo** elements do not reference are considered to be authentication token generators. Each authentication token generator must have a unique **valuetype** element. The following example displays an example of a generator for an X.509 protection token:

```
application.securityoutboundbindingconfig.tokengenerator_0.name=gen_signtgen
application.securityoutboundbindingconfig.tokengenerator_0.classname=com.ibm.ws.wssecurity.wssapi.token
.impl.CommonTokenGenerator
application.securityoutboundbindingconfig.tokengenerator_0.valuetype.uri=
application.securityoutboundbindingconfig.tokengenerator_0.valuetype.localname=http://docs.oasis-open.org
/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.classname=com.ibm.websphere.wssecurity
.callbackhandler.X509GenerateCallbackHandler
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key.alias=soaprequester
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key.keypass={xor}PDM2OjEr
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.key.name=CN=SOAPRequester,
OU=TRL, O=IBM, ST=Kanagawa, C=JP
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.path=${USER_INSTALL_ROOT}
/etc/ws-security/samples/dsig-sender.ks
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.storepass={xor}PDM2OjEr
application.securityoutboundbindingconfig.tokengenerator_0.callbackhandler.keystore.type=JKS
application.securityoutboundbindingconfig.tokengenerator_0.jaasconfig.configname=system.wss.generate.x509
```

The following example displays a generator for a username authentication token:

```
application.securityoutboundbindingconfig.tokengenerator_1.name=gen_usernametoken
application.securityoutboundbindingconfig.tokengenerator_1.classname=com.ibm.ws.wssecurity
.wssapi.token.impl.CommonTokenGenerator
application.securityoutboundbindingconfig.tokengenerator_1.valuetype.uri=
application.securityoutboundbindingconfig.tokengenerator_1.valuetype.localname=http://docs
.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.classname=com.ibm
.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.basicAuth.userid=user1
application.securityoutboundbindingconfig.tokengenerator_1.callbackhandler.basicAuth.password=myPassword
application.securityoutboundbindingconfig.tokengenerator_1.securityTokenReference.reference=request:uname_token
application.securityoutboundbindingconfig.tokengenerator_1.jaasconfig.configname=system.wss.generate.unt
```

tokenconsumer element

In the default bindings, the **tokenconsumer** elements that the **signinginfo** or **encryptioninfo** elements do not reference are authentication token consumers. Each authentication token consumer must have a unique **valuetype** element. The following example displays the format for a set of **tokenconsumer** elements:

```
application.securityinboundbindingconfig.tokenconsumer_0.name=con_unametoken
application.securityinboundbindingconfig.tokenconsumer_0.classname=com.ibm.ws.wssecurity.wssapi
.token.impl.CommonTokenConsumer
application.securityinboundbindingconfig.tokenconsumer_0.valuetype.localname=http://docs.oasis-open.org
/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
application.securityinboundbindingconfig.tokenconsumer_0.valuetype.uri=
application.securityinboundbindingconfig.tokenconsumer_0.callbackhandler.classname=com.ibm.websphere
.wssecurity.callbackhandler.UNTConsumeCallbackHandler
application.securityinboundbindingconfig.tokenconsumer_0.jaasconfig.configname=system.wss.consume.unt
application.securityinboundbindingconfig.tokenconsumer_0.securitytokenreference.reference=request:uname_token
```

actor element

Defines the actor uniform resource identifier (URI) to be included in the WSSecurity headers of a generated message, as displayed by the following example:

```
application.securityinboundbindingconfig.actor=http://myActor.com
application.securityoutboundbindingconfig.actor=http://myActor.com
```

certstorelist element

Defines certificate store configurations and signing information, as displayed by the following example:

```
application.securityinboundbindingconfig.certstorelist.collectioncertstores_0
.name=DigSigCertStore
application.securityinboundbindingconfig.certstorelist.collectioncertstores_0
.provider=IBMCertPath
application.securityinboundbindingconfig.certstorelist.collectioncertstores_0
.x509certificates_0.path=${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer
```

keyinfo element

Defines key information for signing and encryption configurations, as displayed by the following example:

```
application.securityinboundbindingconfig.keyinfo_0.classname=com.ibm.ws.wssecurity.wssapi
.CommonContentConsumer
application.securityinboundbindingconfig.keyinfo_0.name=con_signkeyinfo
application.securityinboundbindingconfig.keyinfo_0.tokenreference.reference=con_tcon
application.securityinboundbindingconfig.keyinfo_0.type=STRREF
```

trustanchor property

Defines configuration information that is used to validate the trust of the signer certificate, as displayed by the following example:

```
application.securityinboundbindingconfig.trustanchor_0.keystore.path=${USER_INSTALL_ROOT}
/etc/ws-security/samples/dsig-receiver.ks
application.securityinboundbindingconfig.trustanchor_0.keystore.storepass={xor}LDotKTot
application.securityinboundbindingconfig.trustanchor_0.keystore.type=JKS
application.securityinboundbindingconfig.trustanchor_0.name=DigSigTrustAnchor
```

timestampexpires element

Defines an expiration date for the configuration, as displayed by the following example:

```
application.securityoutboundbindingconfig.timestampexpires.expires=5
```

application.securityinboundbindingconfig.caller_X.order

Specifies the order for a caller when using wsadmin scripts, where X is the unique string that identifies the instance of the caller:

```
-attributes [[application.securityinboundbindingconfig.caller_0.order 2]]
```

setPolicyType and setBinding command examples

Use the previous reference information with the setPolicyType and setBinding commands to modify your policy and binding configuration data.

Note: The administrative console command assistance provides incorrect Jython syntax for the setPolicyType command. The XPath expression for the response message part protection of the Username WSSecurity policy set contains single quotes (') within each XPath property value, which Jython does not support. To fix the command from the administrative console command assistance, add a backslash character (\) before each single quote to escape the single quote.

The following example uses the setBinding command to set the enabled and provides properties for the myCustomSecurityPS custom policy set, which contains a ReliableMessaging policy:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType WSSecurity
 -attributes [[application.securityinboundbindingconfig.caller_0.order 2][inResponsewithSSL:configAlias NodeDefaultSSLSettings]
[inResponsewithSSL:config properties_directory/ssl.client.props][outAsyncResponsewithSSL:configFile properties_directory/ssl.client.props]
[outAsyncResponsewithSSL:configAlias NodeDefaultSSLSetings][outRequestwithSSL:configFile properties_directory/ssl.client.props]
[outRequestwithSSL:configAlias NodeDefaultSSLSettings]]]')
```

The following setPolicyType command enables the WSSecurity policy and creates a signature protection assertion:

```
AdminTask.setPolicyType('-policySet myPolicySet -policyType WSSecurity -attributes "[[enabled true][provides
Some_amount_of_security][SignatureProtection.request:app_signparts.SignedElements.XPath_0 SignatureProtectionV2]]"')
```

The following setBinding command specifies key information for a server-specific binding:

```
AdminTask.setBinding('-policyType WSSecurity -bindingLocation "[[server server1][node node01]]"
-attributes "[[application.securityinboundbindingconfig.keyinfo_0.name dec_server_keyinfo]
[application.securityinboundbindingconfig.keyinfo_0.classname com.ibm.ws.wssecurity.wssapi.CommonContentGenerator]
[application.securityinboundbindingconfig.keyinfo_0.type STRREF]]"')
```

The following setBinding command specifies key information for an attachment-specific binding:

```
AdminTask.setBinding('-policyType WSSecurity -bindingLocation "[[application PolicySet][attachmentId 999]]"
-attributes "[[application.securityinboundbindingconfig.keyinfo_0.name dec_app_keyinfo]
[application.securityinboundbindingconfig.keyinfo_0.classname com.ibm.ws.wssecurity.wssapi.CommonContentGenerator]
[application.securityinboundbindingconfig.keyinfo_0.type STRREF]]" -attachmentType application
 -bindingName myBindingName')
```

The following setBinding command specifies trust anchor information for a cell-wide binding:

```
AdminTask.setBinding('-policyType WSSecurity -bindingLocation "" -attributes
"[application.securityinboundbindingconfig.trustanchor_0.name DigSigTrustAnchor2]"')
```

WSReliableMessaging policy and binding properties

Use the **attributes** parameter for the setPolicyType and setBinding commands to specify additional configuration information for the ReliableMessaging policy and policy set binding. The WSReliableMessaging quality of service (QoS) is only available for application policy sets.

WSReliableMessaging is an interoperability standard for the reliable transmission of messages between two endpoints. Use WSReliableMessaging to secure and verify transactions when using Web services between businesses.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

WSReliableMessaging policy properties

Configure the WSReliableMessaging policy by specifying the following properties with the setPolicyType command:

specLevel

Choose the WS-ReliableMessaging standard to use for reliable transmission of your messages. The WS-ReliableMessaging specification Version 1.1 is the default value. Use the following information to choose a specification level:

- Specify `1.0` as the value for the specLevel attribute to use the WS-ReliableMessaging specification Version 1.0, February 2005 specification level.
- Specify `1.1` as the value for the specLevel attribute to use the OASIS WS-ReliableMessaging specification Version 1.1, August 2006 specification level.

The following example code sets the specLevel property to the OASIS WS-ReliableMessaging specification Version 1.1, August 2006:

```
AdminTask.setPolicyType('[-policySet "CustomWSReliableMessaging"  -policyType
 WSReliableMessaging -attributes "[[specLevel 1.1]]"]')
```

inOrderDelivery

Specifies whether to process messages in the order that they are received. If you use the inOrderDelivery property, then inbound messages might be queued while waiting for earlier messages.

The following example code enables the inOrderDelivery property:

```
AdminTask.setPolicyType('[-policySet "CustomWSReliableMessaging" -policyType WSReliableMessaging -attributes "[[inOrderDelivery true]]"]')
```

qualityOfService

Specifies the quality of the WSReliableMessaging service to use. Define one of the following three values for the qualityOfService attribute:

- unmanagedNonPersistent

  This setting tolerates network and remote system failures. The unmanagedNonPersistent quality of service is non-transactional. With this setting configured, messages are lost if a server fails. This quality of service is supported for all environments only if the environment is configured as a Web service requester.
- managedNonPersistent

  This setting tolerates system, network, and remote system failures. However, the message state is discarded when the messaging engine restarts. The managedNonPersistent quality of service is non-transactional. This setting prevents message loss if a server fails. However, messages are lost if the messaging engine fails. Managed and thin client applications cannot use this quality of service.
- managedPersistent

  This setting tolerates system, network, and remote system failures. With this setting, messages are processed within transactions, persisted at the Web service requester and provider. Messages can be recovered if a server fails. Messages that are not successfully transmitted at the time of failure continue when the messaging engine or application restarts. Managed and thin client applications cannot use this quality of service.

The following example sets the qualityOfService property as unmanaged nonpersistent:

```
AdminTask.setPolicyType('[-policySet "CustomWSReliableMessaging" -policyType
 WSReliableMessaging -attributes "[[qualityOfService unmanagedNonPersistent]]"]')
```

The following example uses the setPolicyType command to set a value for each policy property:

```
AdminTask.setPolicyType('[-policySet "CustomWSReliableMessaging" -policyType
 WSReliableMessaging -attributes "[[specLevel 1.1][inOrderDelivery true][qualityOfService
 unmanagedNonPersistent]]"]')
```

WSReliableMessaging binding configuration attributes

If you set the qualityOfService policy property to managedNonPersistent or managedPersistent, configure the WSReliableMessaging binding by specifying values for the following properties with the setBinding command:

busName

The name of the service integration bus that contains the messaging engine to use for the managedNonPersistent or managedPersistent Quality of Service options.

The following example sets the busName property as `myBus`:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType
 WSReliableMessaging -attributes "[[busName myBus]]"]')
```

messagingEngineName

The name of the messaging engine to use for the managedNonPersistent or managedPersistent quality of service options.

The following example sets the messagingEngineName property as `messagingEngine001`:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType
 WSReliableMessaging -attributes "[[messageEngineName messageEngine001]]"]')
```

The following code example demonstrates how to use the **setBinding** command to set values for each binding attribute:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType
 WSReliableMessaging -attributes "[[busName myBus][messageEngineName messageEngine001]]"]')
```

WSAddressing binding properties

Use the -attributes parameter for the setBinding command to enable or disable workload management for the WSAddressing binding. Application and system policy sets use the WSAddressing policy and binding.

WSAddressing is an interoperability standard that you can use to create endpoint references that you can distribute across firewalls and intermediary nodes. For more information, see the W3C Candidate Recommendation (CR) versions of the WS-Addressing core and SOAP specifications.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

WSAddressing binding properties

Configure the WSAddressing policy by specifying the following property with the setBinding command:

preventWLM

Specifies whether to prevent workload management for references to endpoints that were created by the application programming interface (API) in a cluster environment. Messages that target Endpoint References (EPRs) within a cluster environment are workload managed by default.

Preventing workload management routes messages that target EPRs to the node or server on which the EPR was created. You might disable workload management if the endpoint maintains the in-memory state, which has not been replicated across other nodes or servers within the cluster.
For example, the following command prevents workload management for a cell-wide general binding, from the WSAddressing policy.

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType WSAddressing -attributes "[preventWLM true]"]')
```

SSLTransport policy and binding properties

Use the -attributes parameter for the setPolicyType and setBinding commands to specify additional configuration information for the SSLTransport policy and policy set binding. Application and system policy sets can use the SSLTransport policy and binding.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the SystemOut.log file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the SystemOut.log file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

SSLTransport policy properties

Use the SSLTransport policy to ensure message security.

Configure the SSLTransport policy by specifying the following properties with the setPolicyType command:

outRequestSSLenabled
Specifies whether to enable the SSL security transport for outbound service requests.

outAsyncResponseSSLenabled
Specifies whether to enable the SSL security transport for asynchronous service responses.

inResponseSSLenabled
Specifies whether to enable the SSL security transport for inbound service responses.

The following setPolicyType command example sets values for all SSLTransport policy properties:

```
AdminTask.setPolicyType('[-policySet "WSHTTPS default" -policyType SSLTransport
-attributes "[[inReponseSSLenabled yes][outAsyncResponseSSLenabled yes][outRequestSSLenabled
 yes]]"]')
```

SSLTransport binding properties

Use the SSLTransport policy type to ensure message security.

Configure the SSLTransport binding by specifying the following properties using the setBinding command:

outRequestwithSSL:configFile
outRequestwithSSL:configAlias
If you enable SSL outbound service requests, then these two attributes define the specific SSL security transport binding and location. The default value for the outRequestwithSSL:configFile attribute is the location of the ssl.client.props file. The default value for the outRequestwithSSL:configAlias attribute is `NodeDefaultSSLSettings`.

outAsyncResponsewithSSL:configFile
outAsyncResponsewithSSL:configAlias
If you enable SSL asynchronous service responses, then these two attributes define the specific SSL security transport binding and location. The default value for the outAsyncRequestwithSSL:configFile attribute is the location of the ssl.client.props file. The default value for the outAsyncRequestwithSSL:configAlias attribute is `NodeDefaultSSLSettings`.

inResponsewithSSL:configFile
inResponsewithSSL:configAlias
If you enable SSL inbound service responses, then these two attributes define the specific SSL security transport binding and location. The default value for the inResponsewithSSL:configFile attribute is the location of the ssl.client.props file. The default value for the inResponsewithSSL:configAlias property is `NodeDefaultSSLSettings`.

The following setBinding command example sets values for all SSLTransport binding attributes:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName cellWideBinding2 -policyType
 SSLTransport -attributes "[[inResponsewithSSL:configAlias NodeDefaultSSLSettings] [inResponsewithSSL:config
 properties_directory/ssl.client.props][outAsyncResponsewithSSL:configFile properties_directory/ssl.client.props]
[outAsyncResponsewithSSL:configAlias NodeDefaultSSLSetings][outRequestwithSSL:configFile
 properties_directory/ssl.client.props][outRequestwithSSL:configAlias NodeDefaultSSLSettings]]"]')
```

HTTPTransport policy and binding properties

Use the -attributes parameter for the setPolicyType and setBinding commands to specify additional configuration information for the HTTPTransport policy and policy set binding. Application and system policy sets can use the HTTPTransport policy and binding.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

The following sections explain the policy and binding properties to configure:

- HTTPTransport policy properties
- HTTPTransport binding properties

HTTPTransport policy properties

The HTTPTransport policy set can be used for HTTPS, basic authorization, compression, and binary encoding transport methods.

Configure the HTTPTransport policy by specifying the following attributes with the **setPolicyType** command:

protocolVersion
Specifies the version of HTTP to use. The valid version values are `HTTP/1.1` and `HTTP/1.0`.

maintainSession
Specifies whether the HTTP session is enabled when a message is sent. The valid values are `yes` or `no`.

chunkTransferEnc
Specifies whether to enable chunked transfer encoding. The valid values are `yes` or `no`.

sendExpectHeader
Specifies whether to send an expect 100-request header. The valid values are `yes` or `no`.

compressRequest:name
Specifies whether to compress the request. The valid values are `gzip`, `x-gzip`, `deflate`, or `none`.

compressResponse:name
Specifies whether to compress the response. The valid values are `gzip`, `x-gzip`, `deflate`, or `none`.

acceptRedirectionURL
Specifies whether to accept URL redirection automatically. The valid values are `yes` or `no`.

messageResendOnce
Specifies if a message can be sent more than once. The valid values are `yes` or `no`.

connectTimeout
Specifies the amount of time, in seconds, before a connection times out when sending a message. Specify an integer value that is greater than zero. If a value of zero or less is specified, the connectTimeout property is set to the default value of 180 seconds. No maximum value is set for this property.

writeTimeout
Specifies the amount of time, in seconds, before the write time out occurs. Specify an integer value. Specify an integer value that is greater than zero. If a value of zero or less is specified, the connectTimeout property is set to the default value of 300 seconds. No maximum value is set for this property.

readTimeout
Specifies the amount of time, in seconds, before the read time out occurs. Specify an integer value. Specify an integer value that is greater than zero. If a value of zero or less is specified, the connectTimeout property is set to the default value of 300 seconds. No maximum value is set for this property.

persistConnection
Specifies whether to use a persistent connection when sending messages. Valid values are `yes` or `no`.

The following setPolicyType example command sets values for each HTTPTransport binding property:

```
AdminTask.setPolicyType('[-policySet "WSHTTPS custom" -policyType HTTPTransport -attributes "[[protocolVersion HTTP/1.1]
[sessionEnable yes][chunkTransferEnc yes][sendExpectHeader yes][compressRequest:name gzip][compressResponse:name
 gzip][acceptRedirectionURL yes][messageResendOnce no][connectTimeout 300][writeTimeout 300]
[readTimeout 300][persistConnection yes]]"]')
```

HTTPTransport binding properties

Configure the HTTPTransport binding by specifying the following attributes with the **setBinding** command:

outAsyncResponseBasicAuth:userid
Specifies the user name for basic authentication of outbound asynchronous responses.

outAsyncResponseBasicAuth:password
Specifies the password for basic authentication of outbound asynchronous responses.

outAsyncResponseProxy:userid
Specifies the user name for the outbound asynchronous service responses proxy.

outAsyncResponseProxy:password
Specifies the password for the outbound asynchronous service responses proxy.

outAsyncResponseProxy:port
Specifies the port number for the outbound asynchronous service responses proxy.

outAsyncResponseProxy:host
Specifies the host name for the outbound asynchronous service responses proxy.

outRequestBasicAuth:userid
Specifies the user name or basic authentication of outbound service requests.

outRequestBasicAuth:password
Specifies the password for basic authentication of outbound service requests.

outRequestProxy:userid
Specifies the user name for the outbound service request proxy.

outRequestProxy:password
Specifies the password for the outbound service request proxy.

outRequestProxy:port
Specifies the port number for the outbound service request proxy.

outRequestProxy:host
Specifies the host name for the outbound service request proxy.

The following **setBinding** example command sets values for each HTTPTransport binding property:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName generalCellWideBind1 -policyType HTTPTransport
-attributes "[[outAsyncResponseBasicAuth:userid myID][outAsyncResponseBasicAuth:password myPW][outAsyncResponseProxy:host hostname]
[outAsyncResponseProxy:port 9060][outAsyncResponseProxy:userid myID][outAsyncResponseProxy:password myPW]
[outRequestBasicAuth:userid myID][outRequestBasicAuth:password myPW][outRequestProxy:userid myID]
[outRequestProxy:password myPW][outRequestProxy:port 9061][outRequestProxy:host hostname]]"]')
```

JMSTransport policy and binding properties

Use the -attributes parameter for the setPolicyType and setBinding commands to specify additional configuration information for the JMSTransport policy and policy set binding. Application policy sets can use the JMSTransport policy and binding.

Use the following commands and parameters in the PolicySetManagement group of the AdminTask object to customize your policy set configuration.

- Use the **attributes** parameter for the getPolicyType and getBinding commands to view the properties for your policy and binding configuration. To get an attribute, pass the property name to the getPolicyType or getBinding command.
- Use the **attributes** parameter for the setPolicyType and setBinding commands to add, update, or remove properties from your policy and binding configurations. To add or update an attribute, specify the property name and value. The setPolicyType and setBinding commands update the value if the attribute exists, or adds the attribute and value if the attribute does not exist. To remove an attribute, specify the value as an empty string (""). The **attributes** parameter accepts a properties object.

Note: If a property name or value supplied with the **attributes** parameter is not valid, then the setPolicyType and setBinding commands fail with an exception. The property that is not valid is logged as an error or warning in the `SystemOut.log` file. However, the command exception might not contain the detailed information for the property that caused the exception. When the setPolicyType and setBinding commands fail, examine the `SystemOut.log` file for any error and warning messages that indicate that the input for the **attributes** parameter contains one or multiple properties that are not valid.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global

security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

The following sections explain the policy and binding properties to configure:

- JMSTransport policy properties
- JMSTransport binding properties

JMSTransport policy properties

Use the JMSTransport policy set to configure JMS transport for applications that use the Java Messaging Service (JMS) to exchange request and response messages.

Configure the JMSTransport policy by specifying the following attributes with the **setPolicyType** command:

requestTimeout
Specifies the request timeout value. The request timeout value is the amount of time, in seconds, that the client waits for a response after sending the request to the server. The default value is `300` seconds. If you specify an integer value of zero or less, the system sets the requestTimeout property to the default value of `300` seconds. No maximum value exists for this property.

allowTransactionalAsyncMessaging
Specifies whether a client uses transactions in one-way or asynchronous two-way requests. The default value for this property is `false`. Set the value of this property to `true` to enable transactional messaging. When enabled, the client runtime exchanges SOAP request and response messages with the server over the JMS transport in a transactional manner if the client operates under a transaction.

The client transaction is used to send the SOAP request message to the destination queue or topic, and the server receives the request message only after the client commits the transaction. Similarly, the server receives the request message under the control of a container-managed transaction and sends the reply message, if applicable, back to the client using that same transaction. Then, the client receives the reply message after the server transaction is committed.

The following setPolicyType example command sets values for each JMSTransport binding property:

```
AdminTask.setPolicyType('[-policySet "JMS custom" -policyType JMSTransport
 -attributes "[[requestTimeout 300][allowTransactionalAsynMessaging false]]"]')
```

JMSTransport binding properties

Configure the JMSTransport binding by specifying the following attributes with the **setBinding** command:

outRequestBasicAuth:userid
Specifies the user name or basic authentication of outbound service requests.

outRequestBasicAuth:password
Specifies the password for basic authentication of outbound service requests.

The following **setBinding** example command sets values for each HTTPTransport binding property:

```
AdminTask.setBinding('[-bindingLocation "" -bindingName generalCellWideBind1
 -policyType JMSTransport -attributes "[[outRequestBasicAuth:userid myID] [outRequestBasicAuth:password myPW]]"]')
```

SecureConversation command group for the AdminTask object (Deprecated)

Use this topic as a reference for the commands for the SecureConversation group of the AdminTask object. Use these commands with your administrative scripts to query, update, and remove secure conversation client cache configuration data.

Note: The commands in the SecureConversation command group are deprecated in WebSphere Application Server Version 7.0. Use the commands in the WSSCacheManagement command group to manage WS-Security distributed cache configurations.

Use the following commands in the SecureConversation group to manage your custom and non-custom secure conversation client cache configurations:

querySCClientCacheConfiguration command

The querySCClientCacheConfiguration command lists all non-custom client cache configuration data for WS-SecureConversation.

Target object

None

Required parameters

None.

Optional parameters

None.

Return value

This command returns a list of all non-custom client cache configuration data.

Batch mode example usage

- Using Jython:

```
print AdminTask.querySCClientCacheConfiguration()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.querySCClientCacheConfiguration('-interactive')
```

querySCClientCacheCustomConfiguration command

The querySCClientCacheCustomConfiguration command lists all custom client cache configuration data for WS-SecureConversation.

Target object

None.

Required parameters

None.

Optional parameters

None.

Return value

This command returns a list of all custom client cache configuration data.

Batch mode example usage

- Using Jython:

```
print AdminTask.querySCClientCacheCustomConfiguration()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.querySCClientCacheCustomConfiguration('-interactive')
```

updateSCClientCacheConfiguration command

The updateSCClientCacheConfiguration command sets the cache cushion time in minutes and enables or disables distributed cache.

Target object

None.

Required parameters

None.

Optional parameters

-distributedCache
Specifies whether distributed cache is enabled or disabled. If you set the -distributedCache parameter to `true` when you run the updateSCClientCacheConfiguration command, the system enables distributed cache for the WS-Security runtime. (Boolean, optional)

-minutesInCacheAfterTimeout
Specifies the amount of time, in minutes, that the token remains in the cache after it expires. The token is renewable for this amount of time. (Integer, optional)

-renewIntervalBeforeTimeoutMinutes
Specifies the amount of time, in minutes, that a renew request is allowed before the token expires. (Integer, optional)

Return value

This command returns a success or failure message.

Batch mode example usage

- Using Jython string:
  ```
  AdminTask.updateSCClientCacheConfiguration('-minutesInCacheAfterTimeout 100 -distributedCache true')
  ```
- Using Jython list:
  ```
  AdminTask.updateSCClientCacheConfiguration(['-minutesInCacheAfterTimeout', '100', '-distributedCache', 'true'])
  ```

Interactive mode example usage
- Using Jython:
  ```
  AdminTask.updateSCClientCacheConfiguration('-interactive')
  ```

updateSCClientCacheCustomConfiguration command

The updateSCClientCacheCustomConfiguration command updates custom properties for the secure conversation client cache configuration.

Target object

None.

Required parameters

None.

Optional parameters

-customProperties
The custom properties for the secure conversation client cache configuration. (Properties, optional)

Return value

This command returns a success or failure message.

Batch mode example usage
- Using Jython string:
  ```
  AdminTask.updateSCClientCacheCustomConfiguration('[-customProperties "[ [property2 value2] [property1 value1] ]"]')
  ```
- Using Jython list:
  ```
  AdminTask.updateSCClientCacheCustomConfiguration(['-customProperties', '[ [property2 value2] [property1 value1] ]'])
  ```

Interactive mode example usage
- Using Jython:
  ```
  AdminTask.updateSCClientCacheCustomConfiguration('-interactive')
  ```

deleteSCClientCacheConfigurationCustomProperties command

The deleteSCClientCacheConfigurationCustomProperties command removes specific properties from a custom secure conversation client cache configuration.

Target object

None.

Required parameters

-propertyNames
The names of the properties to delete. (String, required).

Optional parameters

None.

Return value

This command returns a success or failure message.

Batch mode example usage

- Using Jython string:
  `AdminTask.deleteSCClientCacheConfigurationCustomProperties('[-propertyNames [property1,property2]]')`
- Using Jython list:
  `AdminTask.deleteSCClientCacheConfigurationCustomProperties(['-propertyNames', '[property1,property2]'])`

Interactive mode example usage

- Using Jython:
  `AdminTask.deleteSCClientCacheConfigurationCustomProperties('-interactive')`

WSSCacheManagement command group for the AdminTask object

Use this topic as a reference for the commands for the WSSCacheManagement group of the AdminTask object. Use these commands with your administrative scripts to query, update, and remove distributed cache configuration data.

Use the following commands in the WSSCacheManagement group to manage your custom and non-custom distributed cache configurations:

- "deleteWSSDistributedCacheConfigCustomProperties command"
- "queryWSSDistributedCacheConfig command" on page 1104
- "queryWSSDistributedCacheCustomConfig command" on page 1105
- "updateWSSDistributedCacheConfig command" on page 1105
- "updateWSSDistributedCacheCustomConfig command" on page 1106

deleteWSSDistributedCacheConfigCustomProperties command

The deleteWSSDistributedCacheConfigCustomProperties command removes WS-Security distributed cache custom properties.

Target object

None

Required parameters

-propertyNames
Specifies the names of the custom properties to delete from the distributed cache configuration. (String[])

Optional parameters

None.

Return value

This command returns a message that indicates the success or failure of the command.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteWSSDistributedCacheConfigCustomProperties('[-propertyNames [prop1,prop2,prop3]]')
```

- Using Jython list:

```
AdminTask.deleteWSSDistributedCacheConfigCustomProperties([',-'propertyNames', '[prop1,prop2,prop3]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteWSSDistributedCacheConfigCustomProperties('-interactive')
```

queryWSSDistributedCacheConfig command

The queryWSSDistributedCacheConfig command lists the WS-Security distributed cache configuration non-custom properties.

Target object

None.

Required parameters

None.

Optional parameters

None.

Return value

This command returns a properties object that contains the configuration properties and values for the distributed cache configuration. The following table displays the configuration properties that the command returns:

Property	Description
tokenRecovery	Specifies whether token recovery is enabled or disabled. If the tokenRecovery property is set to `true`, the Datasource property specifies the shared data source that is assigned to the distributed cache.
distributedCache	Specifies whether distributed caching is enabled or disabled.
Datasource	Specifies the name of the shared data source that is assigned to the distributed cache if token recovery is enabled.
renewIntervalBeforeTimeoutMinutes	Specifies the amount of time, in minutes, that the client waits before it attempts to renew the token.
synchronousClusterUpdate	Specifies whether the system performs a synchronous update of distributed caches on cluster members. By default, synchronous cluster updating is enabled.
minutesInCacheAfterTimeout	Specifies the amount of time that the token remains in the cache after the token times out.

Batch mode example usage

- Using Jython:

```
print AdminTask.queryWSSDistributedCacheConfig()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.queryWSSDistributedCacheConfig('-interactive')
```

queryWSSDistributedCacheCustomConfig command

The queryWSSDistributedCacheCustomConfig command lists the WS-Security distributed cache configuration custom properties.

Target object

None.

Required parameters

None.

Optional parameters

None.

Return value

This command returns a properties object that contains the name and value pairs that correspond to each custom property.

Batch mode example usage

- Using Jython:

```
AdminTask.queryWSSDistributedCacheCustomConfig()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.queryWSSDistributedCacheCustomConfig('-interactive')
```

updateWSSDistributedCacheConfig command

The updateWSSDistributedCacheConfig command updates the WS-Security distributed cache configuration non-custom properties.

Target object

None.

Required parameters

None.

Optional parameters

-renewIntervalBeforeTimeoutMinutes
Specifies the amount of time, in minutes, that a renew request is allowed before the token expires. (Integer)

-minutesInCacheAfterTimeout
Specifies the amount of time, in minutes, that the token remains in the cache after it expires. The token is renewable for this amount of time. (Integer)

-distributedCache
Specifies whether distributed cache is enabled or disabled. (Boolean)

-synchronousClusterUpdate
Specifies whether the system performs a synchronous update of distributed caches on cluster members. By default, synchronous cluster updating is enabled. Specify `false` to disable synchronous cluster updating. (Boolean)

-tokenRecovery
Specifies whether token recovery is enabled or disabled. If you set the tokenRecovery property to `true`, specify the shared data source to assign to the distributed cache with the -Datasource parameter. (Boolean)

-Datasource
Specifies the name of the shared data source that is assigned to the distributed cache if token recovery is enabled. (String)

Return value

This command returns a success or failure message.

Batch mode example usage

- Using Jython string:
  ```
  AdminTask.updateWSSDistributedCacheConfig('[-customProperties "[ [property2 value2] [property1 value1] ]"]')
  ```
- Using Jython list:
  ```
  AdminTask.updateWSSDistributedCacheConfig(['-customProperties', '[ [property2 value2] [property1 value1] ]'])
  ```

Interactive mode example usage

- Using Jython:
  ```
  AdminTask.updateWSSDistributedCacheConfig('-interactive')
  ```

updateWSSDistributedCacheCustomConfig command

The updateWSSDistributedCacheCustomConfig command updates the WS-Security distributed cache configuration custom properties.

Target object

None.

Required parameters

-customProperties
Specifies the name and value of each custom property to add or update in the WS-Security distributed cache configuration. (java.util.Properties)

Optional parameters

None.

Return value

This command returns a success or failure message.

Batch mode example usage

- Using Jython string:
  ```
  AdminTask.updateWSSDistributedCacheCustomConfig('[-customProperties [[property1 value1][property2 value2]]]')
  ```
- Using Jython list:

```
AdminTask.updateWSSDistributedCacheCustomConfig(['-customProperties', '[[property1 value1][property2 value2]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updateWSSDistributedCacheCustomConfig('-interactive')
```

PolicySetManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage policy set configurations with the wsadmin tool. Use the commands and parameters in the PolicySetManagement group to create, delete, and manage policy set, policy, and policy set attachment configurations.

Before you use the commands in this topic, verify that you are using the most recent version of the wsadmin tool. The policy set management commands that accept a properties object as the value for the **attributes** or **bindingLocation** parameters are not supported on previous versions of the wsadmin tool. For example, the commands do not run on a Version 6.1.0.x node.

Use the following command to manage policy set configurations:

- “listPolicySets ” on page 1108
- “getPolicySet ” on page 1108
- “createPolicySet ” on page 1109
- “copyPolicySet ” on page 1110
- “deletePolicySet ” on page 1111
- “updatePolicySet ” on page 1111
- “validatePolicySet ” on page 1113
- “exportPolicySet ” on page 1114
- “importPolicySet ” on page 1114

Use the following command to manage policy settings:

- “addPolicyType ” on page 1112
- “deletePolicyType ” on page 1112
- “listPolicyTypes ” on page 1115
- “getPolicyType ” on page 1116
- “setPolicyType ” on page 1117
- “getPolicyTypeAttribute ” on page 1118
- “setPolicyTypeAttribute ” on page 1118

Use the following commands to manage policy set attachments:

- “getPolicySetAttachments ” on page 1119
- “createPolicySetAttachment ” on page 1120
- “updatePolicySetAttachment ” on page 1121
- “addToPolicySetAttachment ” on page 1123
- “removeFromPolicySetAttachment ” on page 1124
- “deletePolicySetAttachment ” on page 1125
- “listAttachmentsForPolicySet ” on page 1127
- “listAssetsAttachedToPolicySet” on page 1127
- “deleteAttachmentsForPolicySet ” on page 1128
- “transferAttachmentsForPolicySet ” on page 1129

Use the following commands to manage policy set bindings:

listPolicySets

The listPolicySets command returns a list of all existing policy sets. If administrative security is enabled, each user role can use this command.

Target object

None.

Optional parameters

-policySetType
Specifies the type of policy set. Specify `application` to display application policy sets. Specify `system` to display system policy sets for trust service or WS-MetadataExchange attachments. Specify `system/trust` to display the policy sets for the trust service. Specify `default` to display the default policy sets. The default value for this parameter is `application`. (String, optional)

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Return value

The command returns a list of all existing policy sets. Each entry in the list is the name of a policy set.

Batch mode example usage

- Using Jython string:

```
AdminTask.listPolicySets('[-policySetType system/trust]')
```

- Using Jython list:

```
AdminTask.listPolicySets(['-policySetType', 'system/trust'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listPolicySets('-interactive')
```

getPolicySet

The getPolicySet command returns general attributes, such as description and default indicator, for the specified policy set. If administrative security is enabled, each user role can use this command.

Target object

None.

Required parameters

-policySet
Specifies the policy set name. For a list of all policy set names, use the listPolicySets command. (String, required)

Optional parameters

-isDefaultPolicySet
Specifies whether to display a default policy set. The default value is `false`. (Boolean, optional)

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Return value

The command returns a list of attributes for the specified policy set name.

Batch mode example usage

- Using Jython string:

```
AdminTask.getPolicySet('[-policySet SecureConversation]')
```

- Using Jython list:

```
AdminTask.getPolicySet(['-policySet', 'SecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getPolicySet('-interactive')
```

createPolicySet

The createPolicySet command creates a new policy set. Policies are not created with the policy set. The default indicator is set to `false`.

If administrative security is enabled, you must use the Administrator role to create policy sets.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set. (String, required)

Optional parameters

-description
Adds a description for the policy set. (String, required)

-policySetType
Specifies the type of policy set. When the value is `application`, the command creates application policy sets. When the value is `system`, the command creates a policy set that you can use for trust service or WS-MetadataExchange attachments. When the value is `system/trust`, the command creates a policy set for the trust service. The default value for this parameter is `application`. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.createPolicySet('[-policySet myCustomPS -description  [my new custom policy
 set] -policySetType system/trust]')
```

- Using Jython list:

```
AdminTask.createPolicySet(['-policySet', 'myCustomPS', '-description',  '[my new custom
 policy set]', '-policySetType', 'system/trust'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createPolicySet('-interactive')
```

copyPolicySet

The copyPolicySet command creates a copy of an existing policy set. By default, the policy set attachments are transferred to the new policy set.

If administrative security is enabled, you must use the Administrator role to copy policy sets.

Target object

None.

Required parameters

-sourcePolicySet
Specifies the name of the existing policy set to copy. (String, required)

-newPolicySet
Specifies the name of the new policy set you are creating. (String, required)

-newDescription
Specifies a description for the new policy set. (String, required)

Optional parameters

-transferAttachments
If this parameter is set to `true`, all attachments transfer from the source policy set to the new policy set. The default value is `false`. (Boolean, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.copyPolicySet('[-sourcePolicySet SecureConversation -newPolicySet
 CustomSecureConversation -newDescription [my new copied policy set] -transferAttachments true]')
```

- Using Jython list:

```
AdminTask.copyPolicySet(['-sourcePolicySet', 'SecureConversation', '-newPolicySet',
 'CustomSecureConversation', '-newDescription', '[my new copied policy set]',  '-transferAttachments',
 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.copyPolicySet('-interactive')
```

deletePolicySet

The deletePolicySet command deletes the specified policy set. If attachments exist for the policy set, the command returns a failure message.

If administrative security is enabled, you must use the Administrator role to delete policy sets.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to delete. (String, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.deletePolicySet('[-policySet customSecureConversation]')
```

- Using Jython list:

```
AdminTask.deletePolicySet(['-policySet', 'customSecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deletePolicySet('-interactive')
```

updatePolicySet

The updatePolicySet command enables you to input an attribute list to update the policy set. You can use this command to update all attributes for the policy set, or a subset of attributes.

If administrative security is enabled, you must use the Administrator role to update policy set configurations.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to update. (String, required)

-attributes
Specifies a properties object that contains the attributes to update for the specified policy set. (Properties, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.updatePolicySet('-policySet policySet1 -attributes [[type application]
 [description [my policy set description]]]')
```

- Using Jython list:

```
AdminTask.updatePolicySet(['-policySet', 'policySet1', '-attributes', '[[type
 application] [description [my policy set description]]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updatePolicySet('-interactive')
```

addPolicyType

The addPolicyType command adds a policy with default values for the specified policy set. You must indicate whether to enable or disable the added policy.

If administrative security is enabled, you must use the Administrator role to add policies.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to update. (String, required)

-policyType
Specifies the name of the policy to add to the policy set. (String, required)

-enabled
If this parameter is set to `true`, new policy is enabled in the policy set. If this parameter is set to `false`, the configuration is contained within the policy set but the configuration does not have an effect on the system. (Boolean, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.addPolicyType('[-policySet customPolicySet -policyType WSTransaction
 -enabled true]')
```

- Using Jython list:

```
AdminTask.addPolicyType(['-policySet', 'customPolicySet', '-policyType',
 'WSTransaction', '-enabled', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addPolicyType('-interactive')
```

deletePolicyType

The deletePolicyType command deletes a policy from a policy set.

If administrative security is enabled, you must use the Administrator role to remove policies from your configuration.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to update. (String, required)

-policyType
Specifies the name of the policy to remove from the policy set. (String, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.deletePolicyType('[-policySet customPolicySet -policyType WSTransaction]')
```

- Using Jython list:

```
AdminTask.deletePolicyType(['-policySet', 'customPolicySet', '-policyType',
 'WSTransaction'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deletePolicyType('-interactive')
```

validatePolicySet

The validatePolicySet command validates the policy set configuration.

If administrative security is enabled, you must use the Administrator role to validate policy sets.

Target object

None.

Required parameters

-policySet
Specifies the policy set to update. (String, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.validatePolicySet('[-policySet customSecureConversation]')
```

- Using Jython list:

```
AdminTask.validatePolicySet(['-policySet', 'customSecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.validatePolicySet('-interactive')
```

exportPolicySet

The exportPolicySet command exports a policy set as an archive that can be copied onto a client environment.

If administrative security is enabled, you must use the Administrator role to export policy sets.

Target object

None.

Required parameters

-policySet
Specifies the policy set to export. (String, required)

-pathName
Specifies the path name of the archive file to create. (String, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportPolicySet('[-policySet customSecureConversation -pathName
 C:/IBM/WebSphere/AppServer/PolicySets/customSC.zip]')
```

- Using Jython list:

```
AdminTask.exportPolicySet(['-policySet', 'customSecureConversation;, '-pathName', '
 C:/IBM/WebSphere/AppServer/PolicySets/customSC.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportPolicySet('-interactive')
```

importPolicySet

The importPolicySet command imports a policy set from a compressed archive file or from a selection of default policy sets onto the server environment.

If administrative security is enabled, you must use the Administrator role to import policy sets.

Target object

None.

Optional parameters

-importFile
Specifies the path name of the archive file to import. (String, optional)

-defaultPolicySet
Specifies the name of the default policy set to import. (String, optional)

-policySet
Specifies the name to assign to the new policy set. If you do not specify this parameter, the system uses the original name of the policy set. (String, optional)

-verifyPolicySetType
Specifies that the policy set type to import matches a specific type. Specify `system` or `system/trust` to verify that the policy set to import is a type of system policy set, including trust service policy sets. Specify `application` to verify that the policy set is an application policy set. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.importPolicySet('[-importFile C:/IBM/WebSphere/AppServer/PolicySets/customSC.zip]')
```

- Using Jython list:

```
AdminTask.importPolicySet(['-importFile', 'C:/IBM/WebSphere/AppServer/PolicySets/customSC.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importPolicySet('-interactive')
```

listPolicyTypes

The listPolicyTypes command returns a list of the names of the policies configured on your system. The input parameters allow you to list each policy type configured in the system, the policy types configured in a policy set, or the policy types in a binding.

If administrative security is enabled, each administrative role can list policy types.

Target object

None.

Optional parameters

-policySet
Specifies the name of the policy set to query for policies. If the policy set is not specified, the command lists all policies defined in your configuration. (String, optional)

-bindingLocation
Specifies the location of the binding. This value is cell-wide default binding, server-specific default binding, or attachment-specific binding. Specify the bindingLocation parameter as a properties object following these guidelines:

- For cell-wide default binding, use a null or empty properties.
- For server-specific default binding, specify the node and server names in the properties. The property names are `node` and `server`. Server-specific default bindings are deprecated.
- For attachment-specific binding, specify the application name and attachment ID in the properties. The property names are `application` and `attachmentId`.
- For system bindings, set the `systemType` property as `trustService`.
- For WSNClient binding, specify the bus name, service name, and attachment ID in the properties. The property names are `bus`, `WSNService`, and `attachmentId`.

(Properties, optional)

-attachmentType
Specifies whether the attachment type is an application binding, client binding, trust service binding, or WS-Notification client binding. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-bindingName
Specifies a specific general binding. If you specify this parameter, the system displays policy types in the specific binding. (String, optional)

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Return value

The command returns a list of policy types.

Batch mode example usage

- Using Jython string:

```
AdminTask.listPolicyTypes('[-policySet customSecureConversation]')
```

- Using Jython list:

```
AdminTask.listPolicyTypes(['-policySet', 'customSecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listPolicyTypes('[-interactive]')
```

getPolicyType

The getPolicyType command returns the attributes for a specified policy.

If administrative security is enabled, each administrative role can query attributes for policies.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to query. (String, required)

-policyType
Specifies the name of the policy of interest. (String, required)

Optional parameters

-attributes
Specifies the specific attributes to display. If this parameter is not used, the command returns all attributes for the specified policy. (String[], optional)

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Return value

The command returns a properties object containing the policy attributes.

Batch mode example usage

- Using Jython string:

```
AdminTask.getPolicyType('[-policySet customSecureConversation -policyType
 SecureConversation]')
```

- Using Jython list:

```
AdminTask.getPolicyType(['-policySet', 'customSecureConversation', '-policyType', 'SecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getPolicyType (['-interactive'])
```

setPolicyType

The setPolicyType command updates the attributes of a specified policy.

Note: The administrative console command assistance provides incorrect Jython syntax for the setPolicyType command. The XPath expression for the response message part protection of the Username WSSecurity policy set contains single quotes (') within each XPath property value, which Jython does not support. To fix the command from the administrative console command assistance, add a backslash character (\) before each single quote to escape the single quote.

If administrative security is enabled, you must use the Administrator role to configure policies.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set of interest. (String, required)

-policyType
Specifies the name of the policy of interest. (String, required)

-attributes
Specifies the specific attributes to be updated. The properties could include all of the policy attributes or a subset of attributes. (Properties, required)

Optional parameters

-replace
Indicates whether the new attributes provided from the command replace the existing policy attributes. For policies with complex data, you can remove optional parts of the configuration when necessary. Use this parameter to get all attributes, perform edits, and replace the binding configuration with the edited data. The default value is `false`. (Boolean, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.setPolicyType('[-policySet customSecureConversation -policyType
 SecureConversation -attributes [[type application] [description [my new description]]]]')
```

- Using Jython list:

```
AdminTask.setPolicyType(['-policySet', 'customSecureConversation', '-policyType', 'SecureConversation', '-attributes',
 '[[type application] [description [my new description]]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setPolicyType('-interactive')
```

getPolicyTypeAttribute

The getPolicyTypeAttribute command returns the value for the specified policy attribute.

If administrative security is enabled, each administrative role can query policy type attribute values.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set of interest. (String, required)

-policyType
Specifies the name of the policy of interest. (String, required)

-attributeName
Specifies the name of the attribute of interest. (String, required)

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Optional parameters

-fromDefaultRepository
Specifies whether to use the default repository. (Boolean, optional)

Return value

The command returns a string that contains the value of the specified attribute.

Batch mode example usage

- Using Jython string:

```
AdminTask.getPolicyTypeAttribute('[-policySet customSecureConversation  -policyType
 SecureConversation -attributeName type]')
```

- Using Jython list:

```
AdminTask.getPolicyTypeAttribute(['-policySet', 'customSecureConversation',  '-policyType',
 'SecureConversation', '-attributeName', 'type'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getPolicyTypeAttribute('-interactive')
```

setPolicyTypeAttribute

The setPolicyTypeAttribute command sets the value for the specified policy attribute.

If administrative security is enabled, you must use the Administrator role to configure policy attributes.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set of interest. (String, required)

-policyType
Specifies the name of the policy of interest. (String, required)

-attributeName
Specifies the name of the attribute of interest. (String, required)

-attributeValue
Specifies the value of the attribute of interest. (String, required)

Return value

If the attribute is successfully added to the policy, the command returns the `true` string value.

Batch mode example usage

- Using Jython string:

```
AdminTask.setPolicyTypeAttribute('[-policySet customPolicySet  -policyType
 WSReliableMessaging -attributeName specLevel  -attributeValue 1.0]')
```

- Using Jython list:

```
AdminTask.setPolicyTypeAttribute(['-policySet', 'customPolicySet',  '-policyType',
 'WSReliableMessaging', '-attributeName', 'specLevel',  '-attributeValue', '1.0'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setPolicyTypeAttribute('-interactive')
```

getPolicySetAttachments

The getPolicySetAttachments command lists the properties for all policy set attachments configured in a specified application.

If administrative security is enabled, each administrative role can query for policy set attachments.

Target object

None.

Optional parameters

-applicationName
Specifies the name of the application to query for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required to query for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-expandResources
Provides expanded information that details the attachment properties for each resource. An asterisk (*

) character returns all Web services. This parameter is valid if the value for the -attachmentType parameter is set to `provider` or `client`. (String, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as `client`, and use the -attachmentProperties parameter to specify the `bus` and `WSNService` properties. For system policy set attachments, specify the attachmentType parameter as `provider`, and use the -attachmentProperties parameter to set the `systemType` property value to `trustService`. (Properties, optional)

Return value

The command returns a list of properties for each attachment in the application, including the policy set name, attachment ID, and resource list. If you specify the expandResources parameter, the command returns the resource, attachmentId, policySet, binding, and directAttachment properties. If a resource is not attached to a policy set, then the system only displays the resource property. The binding property only exists if the attachment contains a custom binding.

Batch mode example usage

- Using Jython string:

```
AdminTask.getPolicySetAttachments('[-attachmentType provider -attachmentProperties "[systemType trustService]"]')
```

- Using Jython list:

```
AdminTask.getPolicySetAttachments(['-attachmentType', 'provider', '-attachmentProperties',
 '[systemType trustService]'])
```

Interactive mode example usage

- Using Jython list:

```
AdminTask.getPolicySetAttachments('-interactive')
```

createPolicySetAttachment

The createPolicySetAttachment command creates a new policy set attachment for an application.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to create policy set attachments. If you have access to a specific resource only, you can create policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to create policy set attachments. If you have access to a specific resource only, you can create policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can create policy set attachments for application resources only.
Operator	The Operator role cannot create policy set attachments.
Monitor	The Monitor role cannot create policy set attachments.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set to attach. (String, required)

-resources
Specifies the name of the application resources to attach to the policy set. (String[], required)

Optional parameters

-applicationName
Specifies the name of the application of interest for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-dynamicClient
Set this parameter to `true`, the system will not recognize the client resources. This option specifies that the client resources are not validated. (Boolean, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as `client`, and use the -attachmentProperties parameter to specify the `bus` and `WSNService` properties. For system policy set attachments, specify the attachmentType parameter as `provider`, and use the -attachmentProperties parameter to set the `systemType` property value to `trustService`. (Properties, optional)

Return value

The command returns a string with the ID of the new attachment.

Batch mode example usage

- Using Jython string:

```
AdminTask.createPolicySetAttachment('[-policySet policyset1 -resources "WebService:/" -applicationName WebService -attachmentType provider]')
```

- Using Jython list:

```
AdminTask.createPolicySetAttachment(['-policySet', 'policyset1', '-resources', '"WebService:/"', '-applicationName', 'WebService',
 '-attachmentType', 'provider'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createPolicySetAttachment('-interactive')
```

updatePolicySetAttachment

The updatePolicySetAttachment command updates the resources that apply to a policy set attachment.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure policy set attachments. If you have access to a specific resource only, you can configure policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to configure policy set attachments. If you have access to a specific resource only, you can configure policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can configure policy set attachments for application resources only.
Operator	The Operator role cannot configure policy set attachments.
Monitor	The Monitor role cannot configure policy set attachments.

Target object

None.

Required parameters

-attachmentId
Specifies the name of the attachment to update. (String, required)

-resources
Specifies the names of the application resources to attach to the policy set. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-dynamicClient
Set this parameter to `true`, the system will not recognize the client resources. This option specifies that the client resources are not validated. (Boolean, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as `client`, and use the -attachmentProperties parameter to specify the `bus` and `WSNService` properties. For system policy set attachments, specify the attachmentType parameter as `provider`, and use the -attachmentProperties parameter to set the `systemType` property value to `trustService`. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.updatePolicySetAttachment('[-attachmentId 123 -resources "WebService:/"]')
```

- Using Jython list:

```
AdminTask.updatePolicySetAttachment(['-attachmentId', '123', '-resources',
 '"WebService:/"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updatePolicySetAttachment ('-interactive')
```

addToPolicySetAttachment

The addToPolicySetAttachment command adds additional resources that apply to a policy set attachment.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to add resources to policy set attachments. If you have access to a specific resource only, you can add resources to policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to add resources to policy set attachments. If you have access to a specific resource only, you can add resources to policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can add resources to policy set attachments for application resources only.
Operator	The Operator role cannot add resources to policy set attachments.
Monitor	The Monitor role cannot add resources to policy set attachments.

Target object

None.

Required parameters

-attachmentId
Specifies the name of the attachment to update. (String, required)

-resources
Specifies the names of the application resources to attach to the policy set. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set

attachments, specify the provider value for the attachmentType parameter and the "[systemType trustService]" value for the -attachmentProperties parameter. For WSNClient attachments, specify the client value for the attachmentType parameter and the bus and WSNService properties with the -attachmentProperties parameter.

-dynamicClient
Set this parameter to true, the system will not recognize the client resources. This option specifies that the client resources are not validated. (Boolean, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as client, and use the -attachmentProperties parameter to specify the bus and WSNService properties. For system policy set attachments, specify the attachmentType parameter as provider, and use the -attachmentProperties parameter to set the systemType property value to trustService. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.addToPolicySetAttachment('[-attachmentId 123 -resources
 "WebService:/webapp1.war:{http://www.ibm.com}myService"]')
```

- Using Jython list:

```
AdminTask.addToPolicySetAttachment(['-attachmentId', '123', '-resources',
 '"WebService:/webapp1.war:{http://www.ibm.com}myService"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addToPolicySetAttachment('-interactive')
```

removeFromPolicySetAttachment

The removeFromPolicySetAttachment command removes resources that apply to a policy set attachment.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to remove resources from policy set attachments. If you have access to a specific resource only, you can remove resources for which you have access.
Configurator	The Configurator role must have cell-wide access to remove resources from policy set attachments. If you have access to a specific resource only, you can remove the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can remove resources from policy set attachments for application resources only.
Operator	The Operator role cannot remove resources from policy set attachments.
Monitor	The Monitor role cannot remove resources from policy set attachments.

Target object

None.

Required parameters

-attachmentId
Specifies the name of the attachment to remove. (String, required)

-resources
Specifies the names of the application resources to attach to the policy set. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as `client`, and use the -attachmentProperties parameter to specify the `bus` and `WSNService` properties. For system policy set attachments, specify the attachmentType parameter as `provider`, and use the -attachmentProperties parameter to set the `systemType` property value to `trustService`. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.removeFromPolicySetAttachment('[-attachmentId 123 -resources
 "WebService:/webapp1.war:{http://www.ibm.com}myService"]')
```

- Using Jython list:

```
AdminTask.removeFromPolicySetAttachment(['-attachmentId', '123', '-resources',
 '"WebService:/webapp1.war:{http://www.ibm.com}myService"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeFromPolicySetAttachment('-interactive')
```

deletePolicySetAttachment

The deletePolicySetAttachment command removes a policy set attachment from an application.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to delete policy set attachments. If you have access to a specific resource only, you can delete policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to delete policy set attachments. If you have access to a specific resource only, you can delete policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can delete policy set attachments for application resources only.
Operator	The Operator role cannot delete policy set attachments.
Monitor	The Monitor role cannot delete policy set attachments.

Target object

None.

Required parameters

-attachmentId
Specifies the name of the attachment to delete. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest for policy set attachments. For application and client attachments, this parameter is required. This parameter is not required for trust service attachments. (String, optional)

-attachmentType
Specifies the type of policy set attachments. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-attachmentProperties
Specifies information that is required to identify the location of the attachment. For WSNClient attachments, specify the attachmentType parameter as `client`, and use the -attachmentProperties parameter to specify the `bus` and `WSNService` properties. For system policy set attachments, specify the attachmentType parameter as `provider`, and use the -attachmentProperties parameter to set the `systemType` property value to `trustService`. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.deletePolicySetAttachment('[-attachmentId 123]')
```

- Using Jython list:

```
AdminTask.deletePolicySetAttachment(['-attachmentId', '123'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deletePolicySetAttachment('-interactive')
```

listAssetsAttachedToPolicySet

The listAssetsAttachedToPolicySet command lists the applications or WS-Notification service clients to which a specific policy set is attached.

If administrative security is enabled, each administrative role can list applications that are attached to policy sets.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set of interest. (String, required)

Optional parameters

-attachmentType
Specifies the type of policy set attachments. The value for this parameter must be `provider`, `client`, `WSNClient`, `WSMex`, or `all`. The default value is `all`. (String, optional)

Return value

The command returns a list of properties that describe each asset. Each properties object contains the `assetType` property, which specifies the type of asset.

Batch mode example usage

- Using Jython string:

```
AdminTask.listAssetsAttachedToPolicySet('[-policySet SecureConversation]')
```

- Using Jython list:

```
AdminTask.listAssetsAttachedToPolicySet(['-policySet', 'SecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAssetsAttachedToPolicySet('-interactive')
```

listAttachmentsForPolicySet

The listAttachmentsForPolicySet command lists the applications to which a specific policy set is attached.

If administrative security is enabled, each administrative role can query for policy set attachments.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set of interest. (String, required)

Optional parameters

-attachmentType
Specifies the type of policy set attachments. The value for this parameter must be `application`, `client`, or `system/trust`. The default value is `application`. (String, optional)

Return value

The command returns a list of application names.

Batch mode example usage

- Using Jython string:

```
AdminTask.listAttachmentsForPolicySet('[-policySet SecureConversation]')
```

- Using Jython list:

```
AdminTask.listAttachmentsForPolicySet(['-policySet', 'SecureConversation'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAttachmentsForPolicySet('-interactive')
```

deleteAttachmentsForPolicySet

The deleteAttachmentsForPolicySet command removes all attachments for a specific policy set.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to delete policy set attachments. If you have access to a specific resource only, you can delete policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to delete policy set attachments. If you have access to a specific resource only, you can delete policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can delete policy set attachments for application resources only.
Operator	The Operator role cannot delete policy set attachments.
Monitor	The Monitor role cannot delete policy set attachments.

Target object

None.

Required parameters

-policySet
Specifies the name of the policy set from which to remove the attachments. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest. The command only deletes attachments for the application of interest if you specify this parameter. (String, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. You can specify values for the `bus` and `WSNService` properties. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAttachmentsForPolicySet('[-policySet customSecureConversation
 -applicationName newApp1]')
```

- Using Jython list:

```
AdminTask.deleteAttachmentsForPolicySet(['-policySet', 'customSecureConversation',
 '-applicationName', 'newApp1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAttachmentsForPolicySet('-interactive')
```

transferAttachmentsForPolicySet

The transferAttachmentsForPolicySet command transfers all attachments from one policy set to another policy set.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to transfer policy set attachments. If you have access to a specific resource only, you can transfer policy set attachments for the resource for which you have access.
Configurator	The Configurator role must have cell-wide access to transfer policy set attachments. If you have access to a specific resource only, you can transfer policy set attachments for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can transfer policy set attachments for application resources only.
Operator	The Operator role cannot transfer policy set attachments.
Monitor	The Monitor role cannot transfer policy set attachments.

Target object

None.

Required parameters

-sourcePolicySet
Specifies the source policy set from which to copy attachments. (String, required)

-destinationPolicySet
Specifies the name of the policy set to which the attachments are copied. (String, required)

Optional parameters

-applicationName
Specifies the name of the application of interest. The command only transfers attachments for the application of interest if you specify this parameter. (String, optional)

-attachmentProperties
Specifies information that is required to identify the location of the attachment. You can specify values for the `bus` and `WSNService` properties. (Properties, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.transferAttachmentsForPolicySet('[-sourcePolicySet SecureConversation
 -destinationPolicySet customSecureConversation -applicationName newApp1]')
```

- Using Jython list:

```
AdminTask.transferAttachmentsForPolicySet(['-sourcePolicySet', 'SecureConversation',
 '-destinationPolicySet', 'customSecureConversation', '-applicationName', 'newApp1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.transferAttachmentsForPolicySet('-interactive')
```

getBinding

The getBinding command returns the binding configuration for a specified policy and scope. You can use the getBinding command to return a list of available custom bindings, which includes bindings that are and are not referenced by attachments.

If administrative security is enabled, each administrative role can query for binding configuration information.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Target object

None.

Required parameters

-policyType
Specifies the policy of interest. (String, required)

-bindingLocation
Specifies the location of the binding. (Properties, required)

Specify the bindingLocation parameter as a properties object following these guidelines:

- For cell-wide general binding or WebSphere Application Server Version 6.1 cell default bindings, specify a null or empty properties.
- For WebSphere Application Server Version 6.1 server-specific default binding, specify the node and server names in the properties. The property names are `node` and `server`. Server-specific default bindings are deprecated.
- For WebSphere Application Server Version 7.0 server default bindings, specify a null or empty properties. Use the bindingName parameter to identify the binding location.
- For attachment-specific, specify the application name and attachment ID in the properties. The property names are `application` and `attachmentId`.
- For WSNClient bindings, specify the bus name, service name, and attachment ID in the properties. The property names are `bus`, `WSNService`, and `attachmentId`. If you specify an asterisk character (*) as the attachment ID, then the command returns the list of binding names that corresponds to the attachment type of interest.
- For system/trust bindings, set the `systemType` property as `trustService`.

Optional parameters

-attachmentType
Specifies the type of policy set attachment. Use this parameter to distinguish between types of attachment custom bindings. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-attributes
Specifies the names of the attributes to return. If this parameter is not specified, the command returns all attributes. (String[], optional)

-bindingName
Specifies the binding name of interest. Specify this parameter to display a general cell-level binding or a custom attachment binding. (String, optional)

Return value

The command returns a properties object that contains the requested configuration attributes for the policy binding.

Batch mode example usage

- Using Jython string:

  The following example returns a list of application bindings:

```
AdminTask.getBinding('[-policyType WSAddressing -attachmentType provider
 -bindingLocation [[application application_name] [attachmentId *]]]')
```

  The following example returns a list of client bindings:

```
AdminTask.getBinding('[-policyType WSAddressing -attachmentType client
 -bindingLocation [[application application_name] [attachmentId *]]]')
```

  The following example returns a list of system bindings:

```
AdminTask.getBinding('[-policyType WSAddressing -attachmentType provider
 -bindingLocation [[systemType trustService] [application application_name] [attachmentId *]]]')
```

- Using Jython list:

```
AdminTask.getBinding(['-policyType', 'WSAddressing', '-bindingLocation', '""'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getBinding('-interactive')
```

setBinding

The setBinding command updates the binding configuration for a specified policy. Use this command to add a server-specific binding, update an attachment to use a custom binding, edit binding attributes, or to remove a binding configuration.

When administrative security is enabled, verify that you use the correct administrative role, as the following table describes:

Administrative role	Authorization
Administrator	The Administrator role must have cell-wide access to configure bindings. If you have access to a specific resource only, you can configure custom bindings for the resource for which you have access. The Administrator role is the only role that can modify binding configurations.
Configurator	The Configurator role must have cell-wide access to assign and unassign bindings. If you have access to a specific resource only, you can assign and unassign bindings for the resource for which you have access.
Deployer	The Deployer role with cell-wide or resource specific access can assign or unassign bindings for application resources only.
Operator	The Operator role cannot configure bindings.
Monitor	The Monitor role cannot configure bindings.

Note: In WebSphere Application Server Version 7.0, the security model is enhanced to a domain-centric security model instead of a server-based security model. The configuration of the default global security (cell) level and default server level bindings has also changed in this version of the product. In the WebSphere Application Server Version 6.1 Feature Pack for Web Services, you can configure one set of default bindings for the cell and optionally configure one set of default bindings for each server. In Version 7.0, you can configure one or more general service provider bindings and one or more general service client bindings. After you have configured general bindings, you can specify which of these bindings is the global default binding. You can also optionally specify general binding that are used as the default for an application server or a security domain.

To support a mixed-cell environment, WebSphere Application Server supports Version 7.0 and Version 6.1 bindings. General cell-level bindings are specific to Version 7.0 Application-specific bindings remain at the version that the application requires. When the user creates an application-specific binding, the application server determines the required binding version to use for application.

Target object

None.

Required parameters

-bindingLocation

Specifies the location of the binding. (Properties, required)

Specify the bindingLocation parameter as a properties object following these guidelines:

- For cell-wide general binding or WebSphere Application Server Version 6.1 cell default bindings, specify a null or empty properties.
- For WebSphere Application Server Version 6.1 server-specific default binding, specify the node and server names in the properties. The property names are `node` and `server`. Server-specific default bindings are deprecated.
- For WebSphere Application Server Version 7.0 server default bindings, specify a null or empty properties. Use the bindingName parameter to identify the binding location.
- For attachment-specific, specify the application name and attachment ID in the properties. The property names are `application` and `attachmentId`.
- For WSNClient bindings, specify the bus name, service name, and attachment ID in the properties. The property names are `bus`, `WSNService`, and `attachmentId`. If you specify an asterisk character (*) as the attachment ID, then the command returns the list of binding names that corresponds to the attachment type of interest.
- For system/trust bindings, set the `systemType` property as `trustService`.

-policyType
Specifies the policy of interest. (String, required)

Optional parameters

-attachmentType
Specifies the type of policy set attachment. Use this parameter to distinguish between types of attachment custom bindings. (String, optional)

Note: The `application` and `system/trust` values for the -attachmentType parameter are deprecated. Specify the `provider` value in place of the `application` value. For system policy set attachments, specify the `provider` value for the attachmentType parameter and the `"[systemType trustService]"` value for the -attachmentProperties parameter. For WSNClient attachments, specify the `client` value for the attachmentType parameter and the `bus` and `WSNService` properties with the -attachmentProperties parameter.

-attributes
Specifies the attribute values to update. This parameter can include all binding attributes for the policy or a subset to update. If the **attributes** parameter is not specified, the command only updates the binding location used by the specified attachment. (Properties, optional)

-bindingName
Specifies the name for the binding. Specify this parameter to assign a new name to an attachment binding or cell-level binding. A name is generated if it is not specified. (String, optional)

-domainName
Specifies the domain name for the binding. This parameter is required when using the command to create and scope a binding to a specific domain other than the administrative security domain. The default value is `global`. (String, optional)

-replace
Specifies whether to replace all of the existing binding attributes with the attributes specified in the command. Use this parameter to remove optional parts of the configuration for policies with complex data. The default value is false. (Boolean, optional)

-remove
Specifies whether to remove a server-specific default binding or to remove a custom binding from an attachment. You cannot remove cell-level default binding. The default value is `false`. (Boolean, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.setBinding('[-policyType WSAddressing -bindingLocation [[application myApplication] [attachmentId 123]] -attributes "[preventWLM false]"
 -attachmentType provider]')
```

- Using Jython list:

```
AdminTask.setBinding(['-policyType', 'WSAddressing', '-bindingLocation', '[[application myApplication] [attachmentId 123]]', '-attributes', '[preventWLM
 false]', '-attachmentType', 'provider'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setBinding('-interactive')
```

getDefaultBindings

The getDefaultBindings command displays the provider and client default bindings if the bindings are set. If the command does not return output, then the system default binding is the current default.

If administrative security is enabled, each administrative role can query for default bindings.

Target object

None.

Optional parameters

-bindingLocation
Specifies the location of the binding. Specify the bindingLocation parameter as a properties object with values for the `node` and `server` properties. (Properties, optional)

-domainName
Specifies the domain name for the binding of interest. This parameter is required if the domain of interest is not in the global security domain and you specified the bindingLocation parameter. The bindingLocation and domainName parameters are mutually exclusive. The default value is `global`. (String, optional)

Return value

The command returns a properties object that contains the names of the provider and client default bindings, if the bindings are set.

Batch mode example usage

- Using Jython string:

```
AdminTask.getDefaultBinding('[-bindingLocation [[node myNode] [server myServer]]]')
```

- Using Jython list:

```
AdminTask.getDefaultBinding(['-bindingLocation', '[[node myNode] [server myServer]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getDefaultBinding('-interactive')
```

getRequiredBindingVersion

The getRequiredBindingVersion command displays the version number of the binding for a specific application.

Target object

None.

Optional parameters

-assetProps
Specifies the name of the application of interest. (Properties, optional)

Return value

The command returns the binding version number as a number, such as `7.0.0.0` or `6.1.0.0`.

Batch mode example usage

- Using Jython string:

```
AdminTask.getRequiredBindingVersion('[-assetProps [[application myApplication]]]')
```

- Using Jython list:

```
AdminTask.getRequiredBindingVersion(['-assetProps', '[[application myApplication]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getRequiredBindingVersion('-interactive')
```

setDefaultBindings

The setDefaultBindings command to set a binding as the default binding.

If administrative security is enabled, you must use the Administrator role with cell-wide access to configure bindings. If you use the Administrator role and do not have cell-wide access, you can only configure bindings on resources for which you have access.

Target object

None.

Required parameters

-defaultBinding
Specifies the names of the default bindings for the provider, client, or both. (Properties, required)

Optional parameters

-bindingLocation
Specifies the location of the binding. Specify the bindingLocation parameter as a properties object with values for the `node` and `server` properties. (Properties, optional)

-domainName
Specifies the domain name for the binding of interest. This parameter is required if the domain of interest is not in the global security domain and you specified the bindingLocation parameter. The bindingLocation and domainName parameters are mutually exclusive. The default value is `global`. (String, optional)

Return value

The command returns a value of `true` if the command successfully sets the default binding.

Batch mode example usage

- Using Jython string:

```
AdminTask.setDefaultBinding('[-bindingName myDefaultBinding -bindingLocation [[node myNode] [server myServer]]]')
```

- Using Jython list:

```
AdminTask.setDefaultBinding(['-bindingName', 'myDefaultBinding', '-bindingLocation', '[[node
 myNode] [server myServer]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setDefaultBinding('-interactive')
```

exportBinding

The exportBinding command export a general, cell-level binding to an archive file. You can copy this file to a client environment or import the archive to a server environment.

If administrative security is enabled, you must use the Administrator role with cell-wide access to export bindings.

Target object

None.

Required parameters

-bindingName
Specifies the name of the binding to assign as the default binding. If you do not specify this parameter, the system specifies the system default as the default binding. (String, required)

-pathName
Specifies the file path for the archive file to create. (String, required)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportBinding('[-bindingName myDefaultBinding -pathName
 C:/IBM/WebSphere/AppServer/PolicySets/Bindings/]')
```

- Using Jython list:

```
AdminTask.exportBinding(['-bindingName', 'myDefaultBinding', '-pathName',
 'C:/IBM/WebSphere/AppServer/PolicySets/Bindings/'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportBinding('-interactive')
```

importBinding

The importBinding command imports a general, cell-level binding from a compressed archive file to a server environment.

If administrative security is enabled, you must use the Administrator role with cell-wide access to import bindings.

Target object

None.

Required parameters

-pathName
Specifies the file path for the archive file to import. (String, required)

Optional parameters

-bindingName
Specifies the name of the binding to assign as the imported binding. If you do not specify this parameter, the system specifies the binding name in the archive file. (String, optional)

-domainName
Specifies a new name of the domain of the binding to import. If you do not specify this parameter, the command uses the domain specified in the archive file. (String, optional)

-verifyBindingType
Verifies that the type of binding to import matches a specific binding type. Specify `provider` to verify that the binding to import is a provider binding, or specify `client` to verify that it is a client binding. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.importBinding('[-bindingName myDefaultBinding -pathName
 C:/IBM/WebSphere/AppServer/PolicySets/Bindings/myBinding.ear]')
```

- Using Jython list:

```
AdminTask.importBinding(['-bindingName', 'myDefaultBinding', '-pathName',
 'C:/IBM/WebSphere/AppServer/PolicySets/Bindings/myBinding.ear'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importBinding('-interactive')
```

copyBinding

The copyBinding command creates a new general, cell-level binding from an existing binding.

If administrative security is enabled, you must use the Administrator role with cell-wide access to copy bindings.

Target object

None.

Required parameters

-sourceBinding
Specifies the name of the existing binding that the system uses to create the new binding. (String, required)

-newBinding
Specifies the name of the binding to create. (String, required)

Optional parameters

-newDescription
Specifies the description text for the new binding. (String, optional)

-domainName
Specifies the domain name for the binding. This parameter is only required if you scope the binding to a domain other than the domain of the source binding. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.copyBinding('[-sourceBinding mySourceBinding -newBinding mySourceCopyBinding]')
```

- Using Jython list:

```
AdminTask.copyBinding(['-sourceBinding', 'mySourceBinding', '-newBinding',
 'mySourceCopyBinding'])
```

Interactive mode example usage

- Using Jython list:

```
AdminTask.copyBinding('-interactive')
```

upgradeBindings

The upgradeBindings command upgrades application bindings for a specific asset to the latest version.

If administrative security is enabled, you must use the Administrator role with cell-wide access to import bindings.

Target object

None.

Required parameters

-assetProps
Specifies the name of the asset of interest. Specify the name of the application as the value for the **application** property. (Properties, required)

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.upgradeBindings('[-assetProps [[application myApplication]]]')
```

- Using Jython list:

```
AdminTask.upgradeBindings(['-assetProps', '[[application myApplication]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.upgradeBindings('-interactive')
```

WS-Policy commands for the AdminTask object

You can use the Jython or Jacl scripting languages to manage WS-Policy settings for Web service resources with the wsadmin tool. You can view or manage how a service provider shares its policies, and how a service client obtains and applies the policies of a service provider.

To run these commands, use the AdminTask object of the wsadmin scripting client. Each command acts on multiple objects in one operation. The commands are provided to allow you to make the most commonly-required types of update in a consistent manner, where modifying the underlying objects directly would be error-prone.

The wsadmin scripting client is run from Qshell. For more information, see the topic "Configure Qshell to run WebSphere Application Server scripts".

These commands are valid only when they are used with WebSphere Application Server Version 7 and later application servers. Do not use them with earlier versions.

The commands to manage WS-Policy settings for Web service resources are part of the PolicySetManagement command group for the AdminTask object.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using these commands, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

The commands that are listed as subtopics are available to manage WS-Policy settings in the PolicySetManagement group of the AdminTask object.

The following commands are available to manage WS-Policy settings in the PolicySetManagement group of the AdminTask object:

- getProviderPolicySharingInfo command
- setProviderPolicySharingInfo command
- getClientDynamicPolicyControl command
- setClientDynamicPolicyControl command

getProviderPolicySharingInfo command

Use the getProviderPolicySharingInfo command to find out whether an application or service that is a Web service provider can share its policy configuration, and list the properties that apply to sharing that configuration.

To run the command, use the AdminTask object of the wsadmin scripting client.

The wsadmin scripting client is run from Qshell. For more information, see the topic "Configure Qshell to run WebSphere Application Server scripts".

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the getProviderPolicySharingInfo command to find out how a Web services application, or a service in a Web services application, shares its policy configuration with clients, service registries, or services that support the WS-Policy specification. The policy configuration is shared in WS-PolicyAttachments format.

The command returns properties that show whether the policy configuration of the resource can be shared with clients through a WS-MetadataExchange request or through Web Service Description Language (WSDL) that is obtained by a ?WSDL HTTP Get request.

Target object

None.

Required parameters

-applicationName
The name of the application for which you want to find out how it shares its policy configuration. The application must be a service provider. (String)

Optional parameters

-resource
The name of the resource for which you want to find out how it shares its policy configuration. If you specify this parameter, only the properties for that resource are returned. To retrieve information for the application, specify `WebService:/`. Alternatively, you can specify a service, endpoint or operation. However, policy sets are attached only at the application or service level, so the properties returned for an endpoint or operation are the settings that are inherited from the service. (String)

Return value

Returns a list of properties that include the resource name and that show whether the policy configuration of the resource can be shared. The following properties can be returned:

wsMexPolicySetName
The name of the policy set that specifies message-level security when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the sharePolicyMethods property is wsMex and a policy set to provide message-level security was specified.

wsMexPolicySetBinding
The name of the binding that is applied when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the sharePolicyMethods property is wsMex and a binding to provide message-level security was specified.

resource
The resource that you specified.

directSetting
How the properties apply to the resource. Valid values for this property are:

true
The properties apply directly to the resource.

false
The properties are inherited from the parent application or service.

sharePolicyMethods
How the policy configuration of the resource can be shared. Valid values for this property are:

httpGet
The resource shares its policy configuration through an HTTP Get request.

wsMex
The resource shares its policy configuration through a WS-MetadataExchange request.

Example

The following command displays the policy sharing configuration properties for the EchoService service in the WSSampleServices application. The provider is configured to share its policy through an HTTP Get request, and a WS-MetadataExchange request with message-level security. Message-level security for the WS-MetadataExchange request is provided using the SystemWSSecurityDefault policy set and the "Provider sample" general binding.

```
AdminTask.getProviderPolicySharingInfo(['-applicationName', 'WSSampleServices',
'-resource', 'WebService:/SampleServicesSei.war:{http://example_path/}EchoService'])
.
.
[ [wsMexPolicySetName SystemWSSecurityDefault] [wsMexPolicySetBinding [Provider sample]]
[resource WebService:/SampleServicesSei.war:{http://example_path/}EchoService/]
[directSetting true] [sharePolicyMethods [httpGet wsMex]] ]
```

setProviderPolicySharingInfo command

Use the setProviderPolicySharingInfo command to set how an application or service that is a Web service provider can share its policy configuration with other clients, service registries, or services that support the WS-Policy specification. You can set or remove this information about how a provider policy is shared.

To run the command, use the AdminTask object of the wsadmin scripting client.

The wsadmin scripting client is run from Qshell. For more information, see the topic "Configure Qshell to run WebSphere Application Server scripts".

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the setProviderPolicySharingInfo command to set how an application, or a service in an application, shares its policy configuration with clients, service registries, or services that support the WS-Policy specification. The policy configuration is shared in WS-PolicyAttachments format.

The policy configuration of the resource can be shared with clients through a WS-MetadataExchange request, through Web Service Description Language (WSDL) exported by a `?WSDL` HTTP Get request, or through both methods.

Target object

None.

Required parameters

-applicationName
The name of the application for which you want to set how the provider policy is shared. (String)

-resource
The name of the resource for which you want to set how the provider policy is shared. For all resources in an application, specify `WebService:/`. For a service in an application, specify `WebService:/`*`module`*`:{`*`namespace`*`}`*`service_name`*. Endpoints or operations inherit the settings of the parent application or service. (String)

Optional parameters

-sharePolicyMethods
Specifies how the policy configuration of the resource can be shared. (String array)

Enter either or both of the following values:

`httpGet`
The resource can share its policy configuration through WSDL that is obtained by a `?WSDL` HTTP Get request.

`wsMex` The resource can share its policy configuration through a WS-MetadataExchange request.

-wsMexProperties
Specifies that message-level security is required for WS-MetadataExchange requests and specifies the settings that provide the message-level security. (Properties)

Enter the following values, following each value with the setting that you require for that value:

`wsMexPolicySetName`
The name of the system policy set that specifies message-level security when the resource shares its policy configuration through a WS-MetadataExchange request. Specify a system policy set that contains only WS-Security policies, only WS-Addressing policies, or both. The default policy set is SystemWSSecurityDefault.

`wsMexPolicySetBinding`
The name of the general binding for the policy set attachment when the resource shares its policy configuration through a WS-MetadataExchange request. Specify a general binding that is scoped to the global domain, or scoped to the security domain of this service. If you do not specify this property, the default binding is used.

This parameter is valid only when you specify `wsMex` for the **sharePolicyMethods** parameter.

-remove

Specifies whether the information about how the provider policy is shared is removed from the resource. (Boolean)

This parameter takes the following values:

true The information about how the provider policy is shared is removed from the resource.

false This value is the default. The information about how the provider policy is shared is not removed from the resource.

Examples

The following example removes the information about how the provider policy is shared from the WSSampleServices application:

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/ -remove true]')
```

The following example enables policy sharing, using WSDL exported by a `?WSDL` HTTP Get request, for the EchoService service in the WSSampleServices application:

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/WSSampleServicesSei.war:{http://example_path/}EchoService
-sharePolicyMethods [httpGet ]]')
```

The following example enables policy sharing, using a WS-MetadataExchange request with message-level security, for the WSSampleServices application. Message level security is provided using the MexPS policy set and the “Client sample” general binding.

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/ -sharePolicyMethods [wsMex ]
-wsMexProperties [ [wsMexPolicySetName [SystemWSSecurityDefault]]
[wsMexPolicySetBinding [Provider sample]] ]]')
```

getClientDynamicPolicyControl command

Use the getClientDynamicPolicyControl command to find out whether an application that is a Web service client obtains the policy configuration of a Web service provider, and to list the properties that apply to obtaining that configuration.

To run the command, use the AdminTask object of the wsadmin scripting client.

The wsadmin scripting client is run from Qshell. For more information, see the topic “Configure Qshell to run WebSphere Application Server scripts”.

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the getClientDynamicPolicyControl command to find out how an application, or a service in an application, obtains the policy configuration of a service provider. The client can obtain the policy configuration of the provider through a Web Services Metadata Exchange (WS-MetadataExchange) request or through an HTTP Get request.

Target object

None.

Required parameters

-applicationName
The name of the application for which you want to find out how it obtains the policy configuration of a service provider. The application must be a service client. (String)

Optional parameters

-resource
The name of the resource for which you want to find out how it obtains the policy configuration of a service provider. If you specify this parameter, only the properties for that resource are returned. To retrieve information for the application, specify `WebService:/`. Alternatively, you can specify a service, endpoint, or operation. However, policy sets are attached only at the application or service level, so the properties returned for an endpoint or operation are the settings that are inherited from the service. (String)

Return value

Returns a list of properties that include the resource name and that show how it obtains the policy configuration of a service provider. The following properties can be returned:

`httpGetTargetURI`
The target URL of the HTTP Get request. This property is returned if the value of the acquireProviderPolicyMethod property is httpGet.

`wsMexPolicySetName`
The name of the policy set that specifies message-level security when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the acquireProviderPolicyMethod property is wsMex and a policy set to provide message-level security was specified.

`wsMexPolicySetBinding`
The name of the binding that is used when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the acquireProviderPolicyMethod property is wsMex and a binding to provide message-level security was specified.

`acquireProviderPolicyMethod`
How the policy configuration of the provider can be obtained. Valid values for this property are:

`wsMex`
The resource can obtain the policy configuration of a service provider through a WS-MetadataExchange request.

`httpGet`
The resource can obtain the policy configuration of a service provider through an HTTP Get request.

resource
The resource that you specified.

directSetting
How the properties apply to the resource. Valid values for this property are:

true
The properties apply directly to the resource.

false
The properties are inherited from the parent application or service.

Examples

The following example displays the properties that control how the EchoService service of the WSPolicyClient application obtains the policy configuration of a service provider. The client is configured to retrieve the provider policy through a WS-MetadataExchange request with message-level security, using the SystemWSSecurityDefault policy set and the “Client sample” general binding.

```
AdminTask.getClientDynamicPolicyControl(['-applicationName', 'WSPolicyClient',
'-resource', 'WebService:/WSPClient.war:{http://example_path/}EchoService'])
.
.
[ [wsMexPolicySetName SystemWSSecurityDefault] [wsMexPolicySetBinding [Client sample]]
[acquireProviderPolicyMethod [wsMex]]
[resource WebService:/WSPClient.war:{http://example_path/}EchoService/]
[directSetting true] ]
```

The following example displays the properties that control how the EchoService service of the WSPolicyClient application obtains the policy configuration of a service provider when the client is configured to retrieve the provider policy through an HTTP Get request.

```
AdminTask.getClientDynamicPolicyControl(['-applicationName', 'WSPolicyClient',
'-resource', 'WebService:/WSPClient.war:{http://example_path/}EchoService'])
.
.
[ [httpGetTargetURI http://example_path/EchoService?wsdl]
[acquireProviderPolicyMethod [httpGet]]
[resource WebService:/WSPClient.war:{http://example_path/}EchoService/]
[directSetting true] ]
```

setClientDynamicPolicyControl command

Use the setClientDynamicPolicyControl command to set how an application that is a Web services client obtains the policy configuration of a Web services provider. You can set, refresh, or remove this information about how a provider policy is obtained.

To run the command, use the AdminTask object of the wsadmin scripting client.

The wsadmin scripting client is run from Qshell. For more information, see the topic “Configure Qshell to run WebSphere Application Server scripts”.

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the setClientDynamicPolicyControl command to set how a client obtains the policy configuration of a service provider.

The client can obtain the policy configuration of the provider through a Web Services Metadata Exchange (WS-MetadataExchange) request or through an HTTP Get request. The service provider must publish its policy in WS-PolicyAttachment format in its Web Service Description Language (WSDL) and the client must be able to support those provider policies.

At run time, the client uses the information to establish a policy configuration that is acceptable to both the client and the service provider.

Target object

An application or service that is a Web services client.

Required parameters

-applicationName
The name of the application for which you want to obtain the policy configuration of the provider. (String)

-resource
The name of the resource for which you want to obtain the policy configuration of the provider. For all resources in an application, specify `WebService:/`. For a service in an application, specify `WebService:/`*`module`*`:{`*`namespace`*`}`*`service_name`*. Endpoints or operations inherit the settings of the parent application or service. (String)

Optional parameters

-acquireProviderPolicyMethod
Specifies how the policy configuration of the provider can be obtained. (String)

Enter one of the following values:

`httpGet`
Obtain the policy configuration of the provider using an HTTP Get request. By default, the HTTP Get request is targeted at the URL for each service endpoint followed by `?WSDL`. If you specify this value for a service, you can use the **httpGetProperties** parameter to change the target of the request.

`wsMex` Obtain the policy configuration of the provider using a WS-MetadataExchange request.

-wsMexProperties
Specifies that message-level security is required for WS-MetadataExchange requests and specifies the settings that provide the message-level security. (Properties)

Enter the following values, following each value with the setting that you require for that value:

`wsMexPolicySetName`
The name of the system policy set that specifies message-level security when the policy configuration of the provider is obtained through a WS-MetadataExchange request. Specify a system policy set that contains only WS-Security policies, only WS-Addressing policies, or both. The default policy set is SystemWSSecurityDefault.

wsMexPolicySetBinding
The name of the general binding for the policy set attachment when the resource shares its policy configuration through a WS-MetadataExchange request. Specify a general binding that is scoped to the global domain, or scoped to the security domain of this service. If you do not specify this property, the default binding is used.

This parameter is valid only when you specify `wsMex` for the **acquireProviderPolicyMethod** parameter.

-httpGetProperties
Specifies the target for an HTTP Get request for a service if you do not want to use the default. (Properties)

Enter the following value, followed by the setting that you require for that value:

httpGetTargetURI
The URL for the service endpoint followed by `?WSDL`.

This parameter is valid only when you specify `httpGet` for the **acquireProviderPolicyMethod** parameter and the resource is a service. Do not use this parameter if the resource is an application.

-remove
Specifies whether to remove the information about how the client obtains the policy configuration of the provider. (Boolean)

This parameter takes the following values:

true Information about how the client obtains the policy configuration of the provider is removed.

false This value is the default. Information about how the client obtains the policy configuration of the provider is not removed.

Examples

The following example removes the information about how the client obtains the policy configuration of the provider from the EchoService service of the WSPolicyClient client application.

```
AdminTask.setClientDynamicPolicyControl('[-applicationName WSPolicyClient
-resource WebService:/WSPolicyClient.war:{http://example_path/}EchoService
-remove true]')
```

The following example configures the EchoService service of the WSPolicyClient client application to obtain the policy configuration of the provider using an HTTP Get request.

```
AdminTask.setClientDynamicPolicyControl('[-applicationName WSPolicyClient
-resource WebService:/WSPolicyClient.war:{http://example_path/}EchoService
-acquireProviderPolicyMethod [httpGet ]
-httpGetProperties [httpGetTargetURI http://example_path?WSDL]]')
```

The following example configures the EchoService service of the WSPolicyClient client application to obtain the the policy configuration of the provider through a WS-MetadataExchange request with message-level security, using the SystemWSSecurityDefault policy set and the “Client sample” general binding.

```
AdminTask.setClientDynamicPolicyControl('[-applicationName WSPolicyClient
-resource WebService:/WSPolicyClient.war:{http://example_path/}EchoService
-acquireProviderPolicyMethod [wsMex ]
-wsMexProperties [ [wsMexPolicySetName [SystemWSSecurityDefault]]
[wsMexPolicySetBinding [Client sample]] ]]')
```

Configuring secure sessions between clients and services using the wsadmin tool

Use the wsadmin tool, which supports the Jython and Jacl scripting language, to edit trust service configurations. Use the STSManagement command group for the AdminTask object to specify details related to secure sessions between clients and target services.

About this task

The trust service uses the secure messaging mechanisms of the Web Services Trust (WS-Trust) specification to define additional extensions for issuing, exchanging, and validating security tokens. Use the STSManagement command group for the AdminTask object to configure the trust service using the wsadmin tool. Complete any of the following tasks using the STSManagement commands:

- Manage token provider configurations.

  Use the wsadmin tool to manage token providers. Customize token providers by defining properties such as token type schema URI, handler factory, cache cushion time, class name, and token timeout. You can also allow or restrict the use of post-dated tokens, distributed cache, and renewable tokens after timeout.
- Query existing token provider configurations.

  Use the wsadmin tool to query the existing trust service token provider configuration.
- Manage endpoint token assignments.

  Use the wsadmin tool to assign, unassign, and modify endpoint token assignments.
- Refresh your configuration changes.

  Use the wsadmin tool to force the trust service to reload the token provider configuration during run time. Complete this action to use new configuration changes before you restart the application server.

What to do next

Use the information center topics for managing token providers using the STSManagement group of commands and the AdminTask object.

Querying the trust service using scripting

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to query the trust service for existing configuration settings. Use the commands in this topic to view current trust service configurations before adding, removing, or editing token provider and endpoint configurations.

About this task

Query your current token provider configurations or endpoint configurations using the STSManagement group of commands. Use the following Jython syntax command examples when writing automation scripts to retrieve configuration attributes and set the output to a variable. Pass the newly set variable to administrative commands in the STSManagement group to automate the editing of token provider and endpoint configurations.

- Use the following command examples to query the trust service for token provider configurations.
  - Determine the local name of the default token provider and set it to the myDefaultTokenType variable.

    The following command sets the *myDefaultTokenType* variable to the local name string for the default token provider:

    ```
    myDefaultTokenType = AdminTask.querySTSDefaultTokenType()
    print myDefaultTokenType
    ```
  - List the local names of each configured token provider.

    The following command sets the *myTokenTypes* variable to an array containing the local names of the configured token providers:

    ```
    myTokenTypes = AdminTask.listSTSConfiguredTokenTypes()
    print myTokenTypes
    ```
  - Display the non-custom properties for the default token provider.

    The following command returns a java.util.Properties instance that contains the values for each non-custom property for the default token provider stored in the *myDefaultTokenType* variable.

    ```
    AdminTask.querySTSTokenTypeConfigurationDefaultProperties(myDefaultTokenType)
    ```

To use this command to query a specific token provider, use the following Jython syntax:

```
AdminTask.querySTSTokenTypeConfigurationDefaultProperties('"Security Context Token"')
```

– Display a properties object containing all custom properties for a token provider configuration.

The following command returns a java.util.Properties instance that contains the values for each of the custom properties for the token provider stored in the *myDefaultTokenType* variable.

```
AdminTask.querySTSTokenTypeConfigurationCustomProperties(myDefaultTokenType)
```

To use this command to query a specific token provider, use the following Jython syntax:

```
AdminTask.querySTSTokenTypeConfigurationCustomProperties('"Security Context Token"')
```

- Use the following command examples to query endpoint target configurations and security constraints for endpoint targets.
  – Display each uniform resource identifier (URI) for each assigned endpoint.

The following command sets the *allMyURIs* variable to an array containing the URIs for each assigned endpoint:

```
allMyURIs = AdminTask.listSTSAssignedEndpoints()
print allMyURIs
```

– Display the token provider that is assigned to a specific endpoint URI.

The following command sets the *myTokenType* variable to the name of the token provider that is assigned to the `http://myserver.mysom.com:9080/Example` endpoint URI:

```
myTokenType = AdminTask.querySTSEndpointTokenType('http://myserver.mysom.com:9080/Example')
print myTokenType
```

What to do next

Use the wsadmin tool to manage and edit token provider and endpoint configurations.

Related tasks

“Configuring secure sessions between clients and services using the wsadmin tool” on page 1147
Use the wsadmin tool, which supports the Jython and Jacl scripting language, to edit trust service configurations. Use the STSManagement command group for the AdminTask object to specify details related to secure sessions between clients and target services.

“Managing existing token providers with scripting”
You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the trust service. Use this topic to modify token provider configuration data, and to add custom properties.

“Associating token providers with endpoint services (targets) using scripting” on page 1154
You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the association of endpoints and tokens. Use this topic to query, assign, and unassign the association of a token provider with an endpoint Uniform Resource Identifier (URI).

Related reference

“STSManagement command group for the AdminTask object” on page 1156
You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the STSManagement group can be used to manage and query trust service token provider configurations and endpoint configurations.

Related information

Trust service targets collection
Use this page to view a list of targets, which are application server service endpoints. You can manage tokens by specifying which token is to be issued when access to a specific endpoint is requested.

Trust service token providers collection
Use this page to view information about or manage token providers for the trust service.

Managing existing token providers with scripting

You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the trust service. Use this topic to modify token provider configuration data, and to add custom properties.

Before you begin

You must have an existing token provider configured in the trust service.

About this task

Use the commands in the STSManagement group of the AdminTask object to modify existing configuration data. This topics includes examples for modifying existing non-custom configuration data.

Modify existing configuration data.
Use the **updateSTSTokenTypeConfiguration** command to update existing properties for a specific token provider configuration. If you specify the -distributedCache parameter, the security context token provider generates a warning and modifies the WS-Security distributed cache configuration. Do not specify a value for the -distributedCache parameter for custom tokens.

1. Determine the token provider configuration to edit.

   Enter the following command to view the list of names of the configured token providers:

   ```
   AdminTask.listSTSConfiguredTokenTypes()
   ```

2. Review the current configuration data for the token provider configuration to edit.

   Enter the following command to view a Properties object containing all non-custom configuration data for the Security Context Token token provider:

```
AdminTask.querySTSTokenTypeConfigurationDefaultProperties('Security Context Token')
```

3. Update the token provider configuration with new configuration data.

   Determine which parameters to update in your configuration, using the following table as a reference:

Parameter	Data type
LocalName Specifies the unique token provider name as the target object of the command.	String, required
-HandlerFactory Specifies the configuration class name, including package information.	String, required
-URI Specifies the unique token type schema URI.	String, required
-lifetimeMinutes Specifies the amount of time, in minutes, that the token is valid.	Integer, optional Default: 120 (minutes) Minimum: 10 (minutes)
-renewalWindowMinutes Specifies the amount of time after the token expires during which the token can be renewed.	Integer, optional Default: 120 (minutes) Minimum: 10 (minutes)
-postdatable Set to `true` to specify that tokens of the token provider are valid at a later time. Tokens can be created with or without a future start time.	Boolean, optional Default: false
-distributedCache (deprecated) Set to `true` to enable distributed cache. If you specify the -distributedCache parameter, the security context token provider generates a warning and modifies the WS-Security distributed cache configuration. Do not specify a value for the -distributedCache parameter for custom tokens.	Boolean, optional Default: false
-renewableAfterExpiration Set to `true` to specify that tokens of the token provider are renewable after expiration.	Boolean, optional Default: false

Parameter	Data type
-tokenCacheFactory (deprecated) Specifies the fully qualified class name for the token provider. The secure conversation token handler class does not recognize this parameter.	String, optional Default: `com.ibm.ws.wssecurity.platform.websphere.trust .server.sts.ext.cache.STSTokenCacheFactoryImpl`

Use the **updateSTSTokenTypeConfiguration** command to update the configuration data for the `Security Context Token` token provider. The following example changes the time that the token is valid from 60 minutes to 100 minutes, disables token renewal after expiration, and enables distributed caching:

```
AdminTask.updateSTSTokenTypeConfiguration('Security Context Token', '[-lifetimeMinutes 100 -renewableAfterExpiration false -distributedCache true]')
```

The command returns a message indicating the success or failure of the operation.

4. Save your configuration changes.

   Use the following command to save your changes:

   ```
   AdminConfig.save()
   ```

5. Reload the modified configuration changes.

   Use the following command to force the trust service to reload your modified configuration without restarting the application server:

   ```
   AdminTask.refreshSTS()
   ```

Related tasks

“Configuring secure sessions between clients and services using the wsadmin tool” on page 1147
Use the wsadmin tool, which supports the Jython and Jacl scripting language, to edit trust service configurations. Use the STSManagement command group for the AdminTask object to specify details related to secure sessions between clients and target services.

“Querying the trust service using scripting” on page 1148
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to query the trust service for existing configuration settings. Use the commands in this topic to view current trust service configurations before adding, removing, or editing token provider and endpoint configurations.

“Adding and removing token provider custom properties using scripting”
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to administer the trust service. Use this topic to set internal system configuration properties for your token provider configuration by adding or removing custom properties.

“Associating token providers with endpoint services (targets) using scripting” on page 1154
You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the association of endpoints and tokens. Use this topic to query, assign, and unassign the association of a token provider with an endpoint Uniform Resource Identifier (URI).

Related reference

Trust service token provider settings
Use this page to modify information for an existing token provider.

“STSManagement command group for the AdminTask object” on page 1156
You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the STSManagement group can be used to manage and query trust service token provider configurations and endpoint configurations.

Adding and removing token provider custom properties using scripting

Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to administer the trust service. Use this topic to set internal system configuration properties for your token provider configuration by adding or removing custom properties.

Before you begin

You must have an existing token provider configured for the trust service.

About this task

Use custom properties to set internal system configuration properties and specify these properties using the customProperties parameter. Custom properties are arbitrary name and value pairs of data, where the name can be a property key or a class implementation, and where the value might be a string or Boolean value. Use this topic and the commands in the STSManagement group for the AdminTask object to add or remove custom properties from your configuration with the Jython scripting language.

- Add new custom properties to a specific token provider configuration.

  Use the **updateSTSTokenTypeConfiguration** command to add or update custom properties to your token provider configuration. Do not use the **updateSTSTokenTypeConfiguration** command to remove custom properties. If you specify the -distributedCache parameter, the security context token provider generates a warning and modifies the WS-Security distributed cache configuration. Do not specify a value for the -distributedCache parameter for custom tokens.

  1. Launch the wsadmin scripting tool using the Jython scripting language.
  2. Determine the token provider configuration to edit.

     Enter the following command to view a list of the names for each configured token provider:

     ```
     AdminTask.listSTSConfiguredTokenTypes()
     ```
  3. Review the configured custom properties for the token provider of interest.

     Enter the following command to view a properties object containing custom configuration data for the *Security Context Token* token provider:

     ```
     AdminTask.querySTSTokenTypeConfigurationCustomProperties('Security Context Token')
     ```
  4. Add custom properties to the token provider configuration.

     Use the **updateSTSTokenTypeConfiguration** command to add the configuration data for the `Security Context Token` token provider. Use the following example to add the `com.ibm.ws.security.webChallengeIfCustomSubjectNotFound` custom property with a value of `false` and the `com.ibm.ws.security.defaultLoginConfig` custom property with a value of `system.DEFAULT` to the configuration:

     ```
     AdminTask.updateSTSTokenTypeConfiguration('Security Context Token', '[-customProperties
     [[com.ibm.ws.security.webChallengeIfCustomSubjectNotFound false]
     [com.ibm.ws.security.defaultLoginConfig system.DEFAULT]] ]')
     ```

     The command returns a message indicating the success or failure of the operation.
  5. Save your configuration changes.

     Use the following command to save your changes:

     ```
     AdminConfig.save()
     ```
  6. Reload the modified configuration changes.

     Use the following command to force the trust service to reload your modified configuration without restarting the application server.

     ```
     AdminTask.refreshSTS()
     ```
- Edit custom properties for a specific token provider configuration.
  1. View configured custom properties for the token provider of interest.

     Enter the following command to view a properties object containing custom configuration data for the *Security Context Token* token provider:

     ```
     AdminTask.querySTSTokenTypeConfigurationCustomProperties('Security Context Token')
     ```
  2. Modify the configuration data for the token provider of interest.

     Use the **updateSTSTokenTypeConfiguration** command to modify the existing configuration data for the *Security Context Token* token provider. This example specifies that the *Security Context*

Token token provider configuration includes the com.ibm.ws.security.webChallengeIfCustomSubjectNotFound custom property with a value of false and the com.ibm.ws.security.defaultLoginConfig custom property with a value of system.DEFAULT. Use the following command to change the value of the com.ibm.ws.security.defaultLoginConfig custom property from system.DEFAULT to system.CUSTOM, and does not change any other configured custom properties:

```
AdminTask.updateSTSTokenTypeConfiguration('Security Context Token', '[-customProperties
[[com.ibm.ws.security.defaultLoginConfig system.CUSTOM]]]')
```

The command returns a message indicating the success or failure of the operation.

3. Save your configuration changes.

   Use the following command to save your changes:

   ```
   AdminConfig.save()
   ```

4. Reload the modified configuration changes.

   Use the following command to force the trust service to reload your modified configuration without restarting the application server:

   ```
   AdminTask.refreshSTS()
   ```

- Remove custom properties from token provider configurations.

1. View configured custom properties for the token provider of interest.

   Enter the following command to view a properties object containing custom configuration data for the *Security Context Token* token provider:

   ```
   AdminTask.querySTSTokenTypeConfigurationCustomProperties('Security Context Token')
   ```

2. Delete the custom property from the token provider configuration.

   Use the **deleteSTSTokenTypeConfigurationCustomProperties** command to delete custom properties from your configuration. Specify the names of the custom properties to remove using the propertyNames parameter. If the specified name does not exist in the configuration, no configuration changes are made. The following command removes the com.ibm.ws.security.webChallengeIfCustomSubjectNotFound and com.ibm.ws.security.defaultLoginConfig custom properties from the *Security Context Token* token provider configuration:

   ```
   AdminTask.deleteSTSTokenTypeConfigurationCustomProperties('Security Context Token',
   '[-propertyNames com.ibm.ws.security.webChallengeIfCustomSubjectNotFound
   com.ibm.ws.security.defaultLoginConfig]')
   ```

   The command returns a message indicating the success or failure of the operation.

3. Save your configuration changes.

   Use the following command to save your changes:

   ```
   AdminConfig.save()
   ```

4. Reload the modified configuration changes.

   Use the following command to force the trust service to reload your modified configuration without restarting the service:

   ```
   AdminTask.refreshSTS()
   ```

Related tasks

“Configuring secure sessions between clients and services using the wsadmin tool” on page 1147
Use the wsadmin tool, which supports the Jython and Jacl scripting language, to edit trust service configurations. Use the STSManagement command group for the AdminTask object to specify details related to secure sessions between clients and target services.

“Querying the trust service using scripting” on page 1148
Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to query the trust service for existing configuration settings. Use the commands in this topic to view current trust service configurations before adding, removing, or editing token provider and endpoint configurations.

“Managing existing token providers with scripting” on page 1149
You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the trust service. Use this topic to modify token provider configuration data, and to add custom properties.

“Associating token providers with endpoint services (targets) using scripting”
You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the association of endpoints and tokens. Use this topic to query, assign, and unassign the association of a token provider with an endpoint Uniform Resource Identifier (URI).

Related reference

Trust service token custom properties
WebSphere Application Server trust service provides several custom properties by default to define the default security context token (SCT).

Trust service token provider settings
Use this page to modify information for an existing token provider.

“STSManagement command group for the AdminTask object” on page 1156
You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the STSManagement group can be used to manage and query trust service token provider configurations and endpoint configurations.

Associating token providers with endpoint services (targets) using scripting

You can use the wsadmin tool, which supports the Jython and Jacl scripting languages, to manage the association of endpoints and tokens. Use this topic to query, assign, and unassign the association of a token provider with an endpoint Uniform Resource Identifier (URI).

Before you begin

Before you can assign and manage endpoint configurations, at least one token provider configuration and a Web service must exist.

About this task

Use the STSManagement group of commands to specify a custom service endpoint Uniform Resource Identifier (URI) and to assign and unassign the association of trust service token providers with endpoint configurations. Complete the steps in this topic to query the trust service for the existing endpoint configuration, associate the default token with an endpoint, and unassociate a token from an endpoint. You can perform these steps in any order.

- Associate a token with a specific endpoint.
  1. View a list of all endpoint URIs that are currently associated with a token provider.

     Before invoking changes on your endpoint configurations, use the following listSTSAssignedEndpoints command to examine your current settings:

     ```
     AdminTask.listSTSAssignedEndpoints()
     ```

If the endpoint of interest is currently associated with a token, do not use the assignSTSEndpointTokenType command. To update the token that is associated with the endpoint, use the updateSTSEndpointTokenType command in the next step.

2. Associate a token with an endpoint.

   Use the assignSTSEndpointTokenType command to specify the token to issue for access to a specific endpoint. You do not need to specify the name of the token provider to assign if the token provider is set as the default configuration. For example, the following command assigns the `Security Context Token` default token to the `http://www.mycompany.com:8080/Ecommerce/Catalog` endpoint URI:

   ```
   AdminTask.assignSTSEndpointTokenType('http://www.mycompany.com:8080/Ecommerce/Catalog')
   ```

   If `Security Context Token` is not the default token provider, use the following command:

   ```
   AdminTask.assignSTSEndpointTokenType('http://www.mycompany.com:8080/Ecommerce/Catalog',
   '-LocalName Security Context Token')
   ```

   The command returns a message indicating the success of the operation.

3. Save your configuration changes.

   Use the following command to save your changes:

   ```
   AdminConfig.save()
   ```

4. Reload the modified configuration changes.

   Use the following command to force the trust service to reload your modified configuration without restarting the application server:

   ```
   AdminTask.refreshSTS()
   ```

- Disassociate a token from an endpoint.

1. Examine the current endpoint configuration.

   Use the listSTSAssignedEndpoints to view a list of each endpoint URI with assigned token providers, as the following example describes:

   ```
   AdminTask.listSTSAssignedEndpoints()
   ```

   The following sample output is displayed:

   ```
   'http://www.mycompany.com:8080/Ecommerce/Catalog'
   ```

2. Choose the endpoint to edit.

   Use the querySTSEndpointTokenType to return the token provider associated with the endpoint of interest. Enter the following command to view the token provider associated with the `http://www.mycompany.com:8080/Ecommerce/Catalog` endpoint URI:

   ```
   AdminTask.querySTSEndpointTokenType('http://www.mycompany.com:8080/Ecommerce/Catalog')
   ```

   The following sample output is displayed:

   ```
   'Security Context Token'
   ```

3. Disassociate the token type from the endpoint.

   Use the **unassignSTSEndpointTokenType** command to disassociate the token provider and endpoint configuration. The following command removes the `Security Context Token` token provider that is associated with the `http://www.mycompany.com:8080/Ecommerce/Catalog` endpoint URI:

   ```
   AdminTask.unassignSTSEndpointTokenType('http://www.mycompany.com:8080/Ecommerce/Catalog',
   '-LocalName Security Context Token')
   ```

   The command returns a message indicating the success of the operation.

4. Save your configuration changes.

   Use the following command to save your changes:

   ```
   AdminConfig.save()
   ```

5. Reload the modified configuration changes.

Use the following command to force the trust service to reload your modified configuration without restarting the service:

```
AdminTask.refreshSTS()
```

STSManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure security with the wsadmin tool. The commands and parameters in the STSManagement group can be used to manage and query trust service token provider configurations and endpoint configurations.

The STSManagement command group contains commands that allow you to configure existing token providers, assign token providers to endpoints, and modify general trust service configuration data. The commands in this group that perform configuration changes require that you execute the save command to commit the changes. No configuration changes are made if an exception is created when executing a command.

Use the following commands to modify and query token provider configurations:

- "createSTSTokenTypeConfiguration "
- "deleteSTSTokenTypeConfigurationCustomProperties " on page 1157
- "listSTSConfiguredTokenTypes " on page 1158
- "querySTSDefaultTokenType " on page 1158
- "querySTSTokenTypeConfigurationDefaultProperties " on page 1159
- "querySTSTokenTypeConfigurationCustomProperties " on page 1160
- "setSTSDefaultTokenType " on page 1160
- "updateSTSTokenTypeConfiguration " on page 1161
- "removeSTSTokenTypeConfiguration" on page 1162

Use the following commands to assign, unassign, and query endpoint configurations:

- "assignSTSEndpointTokenType " on page 1162
- "listSTSAssignedEndpoints " on page 1163
- "listSTSEndpointTokenTypes " on page 1164
- "unassignSTSEndpointTokenType " on page 1164
- "updateSTSEndpointTokenType " on page 1165

Use the following commands to add, edit, delete, and list properties of the trust service:

- "addSTSProperty" on page 1165
- "deleteSTSProperty" on page 1166
- "editSTSProperty" on page 1166
- "listSTSProperties" on page 1167

Use the following command to force the trust service to reload your modified configuration without restarting the application server:

- "refreshSTS " on page 1168

createSTSTokenTypeConfiguration

The createSTSTokenTypeConfiguration command is used to create a token provider configuration.

Target object

Specify the LocalName object, which is used as an identifier for the various configurations. The value for the LocalName object must be unique.

Required parameters

-URI
The URI of the token provider. This value must be unique across all configuration token type URIs. (String, required)

-HandlerFactory
Provide the fully qualified class name of an implementation of the `org.eclipse.higgins.sts.IObjectFactory` interface. (String, required)

Optional parameters

-lifetimeMinutes
Specifies the maximum lifetime to assign to an issued token provider. The default value is 120 minutes. (Integer, optional)

-distributedCache
Specifies whether to enable or disable distributed cache. Specify `true` to enable distributed cache capability. The default value is `false`. If you specify this option, the security context token provider generates a warning and modifies the WS-Security distributed cache configuration. Do not specify a value for this parameter for custom tokens. (Boolean, optional)

-tokenCacheFactory
Specifies the fully qualified class name for the token provider. The secure conversation token handler class does not recognize this parameter. (String, optional).

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.createSTSTokenTypeConfiguration('myTokenType', '[-HandlerFactory
 test.ibm.samples.myTokenType -URI http://ibm.com/tokens/schema/myTokenType]')
```

- Using Jython list:

```
AdminTask.createSTSTokenTypeConfiguration('myTokenType', ['-HandlerFactory',
 'test.ibm.samples.myTokenType', '-URI', 'http://ibm.com/tokens/schema/myTokenType'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createSTSTokenTypeConfiguration('-interactive')
```

deleteSTSTokenTypeConfigurationCustomProperties

The deleteSTSTokenTypeConfigurationCustomProperties command is used to remove custom properties from a token provider configuration.

Target object

Specify the LocalName object of the token provider of interest.

Required parameters

None

Optional parameters

-propertyNames
Specify the names of the custom properties to delete from the configuration. If any of the specified properties do not exist in your configuration, you will receive an error message. (String[], optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteSTSTokenTypeConfigurationCustomProperties('myTokenType', '[-propertyNames
 com.ibm.ws.security.webChallengeIfCustomSubjectNotFound com.ibm.ws.security.defaultLoginConfig]')
```

- Using Jython list:

```
AdminTask.deleteSTSTokenTypeConfigurationCustomProperties('myTokenType', ['-propertyNames',
 'com.ibm.ws.security.webChallengeIfCustomSubjectNotFound com.ibm.ws.security.defaultLoginConfig'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteSTSTokenTypeConfigurationCustomProperties('-interactive')
```

listSTSConfiguredTokenTypes

The listSTSConfiguredTokenTypes command is used to list the local names of all configured token providers.

Target object

None

Required parameters

None

Optional parameters

None

Return value

The command returns the local names of all configured token providers.

Batch mode example usage

- Using Jython:

```
AdminTask.listSTSConfiguredTokenTypes()
```

Interactive mode example usage

- Using Jython:

```
AdminTasklistSTSConfiguredTokenTypes('-interactive')
```

querySTSDefaultTokenType

The querySTSDefaultTokenType command is used to determine the local name of the default token provider.

Target object

None

Required parameters

None

Optional parameters

None

Return value

The command returns the local name of the default token provider.

Batch mode example usage

- Using Jython:

```
AdminTask.querySTSDefaultTokenType()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.querySTSDefaultTokenType('-interactive')
```

querySTSTokenTypeConfigurationDefaultProperties

The querySTSTokenTypeConfigurationDefaultProperties command is used to query the trust service for the non-custom properties of a token provider.

Target object

Specify the LocalName object of the token provider to query.

Required parameters

None

Optional parameters

None

Return value

The command returns a `java.util.Properties` instance which contains the values of the non-custom properties. Non-custom properties include URI, HandlerFactory, lifetimeMinutes, distributedCache, postdatable, renewableAfterExpiration, and renewalWindowMinutes.

Batch mode example usage

- Using Jython:

```
AdminTask.querySTSTokenTypeConfigurationDefaultProperties('TokenType2')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.querySTSTokenTypeConfigurationDefaultProperties('-interactive')
```

querySTSTokenTypeConfigurationCustomProperties

The querySTSTokenTypeConfigurationCustomProperties command is used to query the trust service.

Target object

Specify the LocalName object of the token provider of interest.

Required parameters

None

Optional parameters

None

Return value

The command returns a `java.util.Properties` instance containing the values of the custom properties.

Batch mode example usage

- Using Jython:

```
AdminTask.querySTSTokenTypeConfigurationCustomProperties('TokenType2')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.querySTSTokenTypeConfigurationCustomProperties('-interactive')
```

setSTSDefaultTokenType

The setSTSDefaultTokenType command is used to set the default token provider for the trust service.

Target object

Specify the LocalName object of the token provider as default.

Required parameters

None

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython:

```
AdminTask.setSTSDefaultTokenType('TokenType2')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.setSTSDefaultTokenType('-interactive')
```

updateSTSTokenTypeConfiguration

The updateSTSTokenTypeConfiguration command is used to update configuration data for a token provider. All parameters are optional. The parameters that are specified are updated in the configuration if the property already exists. If the property does not exist, it is added to the configuration. To remove custom properties, use the **deleteSTSTokenTypeConfigurationCustomProperties** command.

Target object

Specify the LocalName object of the token provider of interest.

Required parameters

None

Optional parameters

-URI
The URI of the token provider. This value must be unique across all configuration token type URIs. (String, optional)

-HandlerFactory
Provide the fully qualified class name of an implementation of the `org.eclipse.higgins.sts.utilities.IObjectFactory` interface. (String, optional)

-lifetimeMinutes
The maximum lifetime to assign to an issued token provider. The default value is 120 minutes. (Integer, optional)

-distributedCache
Specifies whether to enable or disable distributed cache. Specify `true` to enable distributed cache capability. The default value is `false`. If you specify this option, the security context token provider generates a warning and modifies the WS-Security distributed cache configuration. Do not specify a value for this parameter for custom tokens. (Boolean, optional)

-postdatable
Set the value of this parameter to `true` to allow tokens of this token provider to be valid starting at a future time. The default value is false. (Boolean, optional)

-renewableAfterExpiration
Set the value of this parameter to `true` to allow tokens of this token provider to be renewable after expiration. The default value is false. (Boolean, optional)

-renewableWindowMinutes
Provide the number of minutes after a token has expired that a token of this token provider can be renewed. If this specified time has elapsed after expiration, then the token will no longer be available for renewal. The default value is 120 minutes. (Integer, optional)

-tokenCacheFactory
Specifies the fully qualified class name for the token provider. The secure conversation token handler class does not recognize this parameter. (String, optional).

-customProperties
Provide any additional custom properties. (java.util.Properties, optional).

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.updateSTSTokenTypeConfiguration('myTokenType', '[-lifetimeMinutes 100
 -renewableAfterExpiration false -distributedCache true]')
```

- Using Jython list:

```
AdminTask.updateSTSTokenTypeConfiguration('myTokenType', ['-lifetimeMinutes', '100', '
 -renewableAfterExpiration', 'false', '-distributedCache', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updateSTSTokenTypeConfiguration('-interactive')
```

removeSTSTokenTypeConfiguration

The removeSTSTokenTypeConfiguration command removes a token provider configuration.

Target object

Specify the LocalName object of the token provider of interest.

Required parameters

None

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython:

```
AdminTask.removeSTSTokenTypeConfiguration('myTokenType')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.removeSTSTokenTypeConfiguration ('-interactive')
```

assignSTSEndpointTokenType

The assignSTSEndpointTokenType command is used to give a token provider when a specific endpoint is accessed.

Target object

Specify the endpointURI object of the endpoint to assign a given token provider. If the specified endpoint has already been assigned a token provider, you will receive an error message.

Required parameters

None

Optional parameters

-LocalName
Specify the local name of the token provider to assign to the specified endpoint. If the token provider configuration does not exist, you will receive an error message. If this parameter is not specified, the default token provider is used. (String, optional)

-issuer
Specify the URI of the issuer that specifies the token provider to issue. This value can be null. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.assignSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/', '[-LocalName
 tokenType1]')
```

- Using Jython list:

```
AdminTask.assignSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/', ['-LocalName',
 'tokenType1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.assignSTSEndpointTokenType ('-interactive')
```

listSTSAssignedEndpoints

The listSTSAssignedEndpoints command is used to list the URIs of assigned endpoints.

Target object

None

Required parameters

None

Optional parameters

None

Return value

The command returns the URIs of all assigned endpoints.

Batch mode example usage

- Using Jython:

```
AdminTask.listSTSAssignedEndpoints()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSTSAssignedEndpoints ('-interactive')
```

listSTSEndpointTokenTypes

The listSTSEndpointTokenTypes command is used to query the Trust Service for the token provider assigned to a specific endpoint.

Target object

Specify the endpointURI object of the endpoint to query. An exception is raised if the specified endpoint has not been assigned a token provider.

Required parameters

None

Optional parameters

None

Return value

The command returns the local name of the token provider assigned to the specified endpoint.

Batch mode example usage

- Using Jython:

```
AdminTask.listSTSEndpointTokenTypes()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSTSEndpointTokenTypes ('-interactive')
```

unassignSTSEndpointTokenType

The unassignSTSEndpointTokenType command is used to unassign an endpoint from its token provider.

Target object

Specify the endpointURI object of the endpoint to unassign from a given token provider. An exception is raised if the specified endpoint has not been assigned a token provider.

Required parameters

-LocalName
Specify the local name of the token provider configuration to unassign from the specified endpoint. (String, required)

Optional parameters

-issuer
Specify the URI of the issuer in the token provider assignment to remove. (String, optional)

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.unassignSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/','[-LocalName
 tokenType2]')
```

- Using Jython list:

```
AdminTask.unassignSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/', ['-LocalName',
 'tokenType2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.unassignSTSEndpointTokenType ('-interactive')
```

updateSTSEndpointTokenType

The updateSTSEndpointTokenType command is used to assign a different token provider to a specified endpoint.

Target object

Specify the endpointURI object of the endpoint to update. An exception is raised if the specified endpoint has not been assigned a token provider.

Required parameters

-LocalName
Specify the local name of the token provider to assign to the specified endpoint. If the token provider configuration does not exist, you will receive an error message. If this parameter is not specified, the default token provider is used. (String, optional)

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.updateSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/', '[-LocalName
 tokenType2]')
```

- Using Jython list:

```
AdminTask.updateSTSEndpointTokenType('www.ibm.tokenservice/Ecommerce/', ['-LocalName',
 'tokenType2'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updateSTSEndpointTokenType('-interactive')
```

addSTSProperty

The addSTSProperty command adds a new property for the trust service.

Target object

Specify a unique name for the new property (string, required).

Required parameters

-propertyValue
Specifies the value of the property to add. (String, required)

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.addSTSProperty('pluginSCTVersion', '[-propertyValue 2.0]')
```

- Using Jython list:

```
AdminTask.addSTSProperty('pluginSCTVersion', ['-propertyValue', '2.0'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addSTSProperty('-interactive')
```

deleteSTSProperty

The deleteSTSProperty command deletes an existing property from the trust service.

Target object

Specify the name of the property to delete.

Required parameters

None

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython:

```
AdminTask.deleteSTSProperty('pluginSCTVersion')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteSTSProperty('-interactive')
```

editSTSProperty

The editSTSProperty command modifies an existing property for the trust service.

Target object

Specify the name of the property to edit. (String, required)

Required parameters

-propertyValue
Specifies the new value for the property of interest. (String, required)

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython string:

```
AdminTask.editSTSProperty('pluginSCTVersion', '[-propertyValue 2.1]')
```

- Using Jython list:

```
AdminTask.editSTSProperty('pluginSCTVersion', ['-propertyValue', '2.1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editSTSProperty('-interactive')
```

listSTSProperties

The listSTSProperties command lists all existing properties and their corresponding values for the trust service.

Target object

None

Required parameters

None

Optional parameters

None

Return value

The command returns a java.util.Properties instance that contains the names and values of the properties.

Batch mode example usage

- Using Jython:

```
AdminTask.listSTSProperties()
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listSTSProperties('-interactive')
```

refreshSTS

The refreshSTS command refreshes your trust service configuration changes without restarting the application server.

Target object

None

Required parameters

None

Optional parameters

None

Return value

The command returns a success or failure message.

Batch mode example usage

- Using Jython:

```
AdminTask.refreshSTS()
```

Chapter 15. Using the Administration Thin Client

With the Administration Thin Client, you can run the wsadmin tool or a standalone administrative Java program with only a couple of JAR files. This reduces the amount of time that it takes for the wsadmin tool to start and improved performance. This information should be used to set up JMX client programs.

Before you begin

Verify that the Java™ Development Kit (JDK) is installed on the Administration Thin Client.

About this task

The Administration Thin Client does not support the installation of SAR files or editing applications that use an external JACC provider, for example, Tivoli Access Manager.

For tracing and logging information for the Administration Thin Client, see the Enabling trace on client and standalone applications article in the *Troubleshooting and support* PDF.

1. Make the Administration Thin Client JAR files available by copying com.ibm.ws.admin.clientXXX.jar from a WebSphere Application Server environment to an environment outside of WebSphere Application Server, for example, *MyThinClient*. The `com.ibm.ws.admin.client_7.0.0.jar` Administration Thin Client JAR file is located in one of the following locations:
   - The *AppServer*/`runtimes` directory.
   - The *AppClient*/`runtimes` directory, if you optionally selected the Administration Thin Client when you installed the application client.

   You must have a special license agreement to download these JAR files.
2. Use the Administration Thin Client JAR files to compile and test administration client programs. For Java applications, you can compile and run the JAR files within a standard Java 2 Platform, Standard Edition environment. For more information see the “Compiling an application in a non-OSGi environment using scripting” on page 1170 topic.
3. Copy the messages directory from the *app_server_root*//`properties` directory to the `C:\MyThinClient\properties` directory.
4. If security is turned on, you will also need the following required files:
   - Copy the `com.ibm.ws.security.crypto.jar` file from either the `AppServer/plugins` directory or the `AppClient/plugins` directory and put it in the *MyThinClient* directory.
   - Copy the `soap.client.props` file from the `AppServer\profiles\`*profileName*`\properties` directory and put it in the *MyThinClient*`\properties` directory. Then, enable the client security by setting the `com.ibm.CORBA.securityEnabled` property to `true`.
   - Copy the `sas.client.props` file from the `AppServer/profiles/`*profileName*`/properties` directory and put it to *MyThinClient*`\properties` directory
   - Copy the `ipc.client.props` file from the `AppServer/profiles/profileName/properties` directory and put it to `/MyThinClient/properties` directory.
   - Copy the `ssl.client.props` file from either the `AppServer\profiles\`*profileName*`/properties` directory or the `AppClient/properties` directory and put it in the *MyThinClient*`\properties` directory. Note that this file contains the `user.root` property. You must modify the value to your thin client directory, for example, *MyThinClient*.
   - Copy the `key.p12` and `trust.p12` files from `AppServer\profiles\`*profileName*`\etc` directory and put it to *MyThinClient*`\etc` directory. To do this, copy the file to your thin client directory or run a script to generate the file. For more information, see the following articles:
     – ssl.client.props client configuration file
     – Interoperating with previous product versions

- retrieveSigners command
- Secure installation for client signer retrieval

5. Launch the Administration Thin Client client or run the wsadmin tool remotely in a Java 2 Platform, Standard Edition environment.
6. Deploy the stand alone JAR files and the administrative client applications and start the applications outside of a WebSphere Application Server environment.

Compiling an application in a non-OSGi environment using scripting

Use the wsadmin tool to use the thin administrative client to compile an application.

About this task

To use the thin administrative client JAR files to compile an application, perform the following steps:

1. Include the `com.ibm.ws.admin.client_7.0.0.jar` JAR file in the CLASSPATH.
2. Compile the application.

Example

For example, the JAR file is located in the *app_server_root*`/runtimes` directory and to compile the ThinAdminClientApplication.java file in the current directory. Use the following to compile the file::

```
export CLASSPATH=app_server_root/runtimes/com.ibm.ws.admin.client_7.0.0.jar:${CLASSPATH}
${JAVA_HOME}/bin/javac ThinAdminClientApplication.java
```

Running the wsadmin tool remotely in a Java 2 Platform, Standard Edition environment

The thin administrative client adds JAR files that support administrative client functions that you can use with IBM Developer Kits For the Java Platform.

About this task

For more information about thin administrative clients, see the Application client functions article in the *Developing and deploying applications* PDF.

Thin administrative clients do not support the installation of SAR files or the editing of applications that use an external JACC provider such as Tivoli Access Manager.

For tracing and logging information for the thin administrative client, see the Enabling trace on client and standalone applications article in the *Troubleshooting and support* PDF.

1. Obtain the thin administrative client JAR file and other required files that are required when security is on from the WebSphere Application Server Network Deployment installation. Refer to the Chapter 15, “Using the Administration Thin Client,” on page 1169 article for details about the files that you need to perform this task.
2. Generate the wsadmin.sh or the wsadmin.bat file from the server machine. This file does not ship with the application client. There is an example of wsadmin.bat step 4.
3. Copy the Java directory from the server installation to your thin client environment.
4. Start the wsadmin tool in a non-OSGi environmnet. The following is an example the wsadmin.bat file:

```
@REM wsadmin launcher @echo off @REM Usage: wsadmin arguments setlocal @REM was home should point
 to whatever directory you decide for your thin client environment set WAS_HOME=c:\MyThinClient set USER_INSTALL_ROOT=%WAS_HOME%
 @REm Java home should point to where you installed java for your thinclient set JAVA_HOME="%WAS_HOME%\java" set
 WAS_LOGGING=-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager -Djava.util.logging.configureByServer=true  if exist
 "%JAVA_HOME%\bin\java.exe" (    set JAVA_EXE="%JAVA_HOME%\bin\java" ) else (    set JAVA_EXE="%JAVA_HOME%\jre\bin\java" )  @REM
 CONSOLE_ENCODING controls the output encoding used for stdout/stderr @REM console - encoding is correct for a console window @REM
 file - encoding is the default file encoding for the system @REM other - the specified encoding is used.  e.g. Cp1252,
 Cp850, SJIS @REM SET CONSOLE_ENCODING=-Dws.output.encoding=console  @REM For debugging the utility itself @REM set
```

```
WAS_DEBUG=-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777  set
CLIENTSOAP=-Dcom.ibm.SOAP.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\soap.client.props set
CLIENTSAS=-Dcom.ibm.CORBA.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\sas.client.props set
CLIENTSSL=-Dcom.ibm.SSL.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\ssl.client.props set
CLIENTIPC=-Dcom.ibm.IPC.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\ipc.client.props  @REM the following are wsadmin property
 @REM you need to change the value to enabled to turn on trace set
wsadminTraceString=-Dcom.ibm.ws.scripting.traceString=com.ibm.*=all=disabled set
wsadminTraceFile=-Dcom.ibm.ws.scripting.traceFile="%USER_INSTALL_ROOT%"\logs\wsadmin.traceout set
wsadminValOut=-Dcom.ibm.ws.scripting.validationOutput="%USER_INSTALL_ROOT%"\logs\wsadmin.valout @REM this will be the server host
that you will connecting to set wsadminHost=-Dcom.ibm.ws.scripting.host=myhost.austin.ibm.com  @REM you need to make sure the
port number is the server SOAP port number you want to connect to, in this example the server SOAP port is 8887 set
wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=SOAP set wsadminPort=-Dcom.ibm.ws.scripting.port=8887  @REM you need to
make sure the port number is the server RMI port number you want to connect to, in this example the server RMI Port is 2815 @REM
set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=RMI @REM set wsadminPort=-Dcom.ibm.ws.scripting.port=2815  @REM you
need to make sure the port number is the server JSR160RMI port number you want to connect to, in this example the server
JSR160RMI Port is 2815 @REM set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=JSR160RMI @REM set
wsadminPort=-Dcom.ibm.ws.scripting.port=2815  @REM you need to make sure the port number is the server IPC port number you want
to connect to, in this example the server IPC Port is 9632 and the host for IPC should be localhost @REM set
wsadminHost=-Dcom.ibm.ws.scripting.ipchost=localhost @REM set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=IPC @REM set
wsadminPort=-Dcom.ibm.ws.scripting.port=9632  @REM specify what language you want to use with wsadmin set
wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jacl @REM set wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jython  set
SHELL=com.ibm.ws.scripting.WasxShell  :prop set WSADMIN_PROPERTIES_PROP= if not defined WSADMIN_PROPERTIES goto workspace set
WSADMIN_PROPERTIES_PROP="-Dcom.ibm.ws.scripting.wsadminprops=%WSADMIN_PROPERTIES%"  :workspace set WORKSPACE_PROPERTIES= if not
defined CONFIG_CONSISTENCY_CHECK goto loop set WORKSPACE_PROPERTIES="-Dconfig_consistency_check=%CONFIG_CONSISTENCY_CHECK%"
:loop if '%1'=='-javaoption' goto javaoption if '%1'==' ' goto runcmd goto nonjavaoption  :javaoption shift set
javaoption=%javaoption% %1 goto again  :nonjavaoption set nonjavaoption=%nonjavaoption% %1  :again shift goto loop   :runcmd  set
C_PATH="%WAS_HOME%\properties;%WAS_HOME%\com.ibm.ws.admin.client_7.0.0.jar;%WAS_HOME%\com.ibm.ws.security.crypto.jar" set
PERFJAVAOPTION=-Xms256m -Xmx256m -Xj9 -Xquickstart  if "%JAASSOAP%"=="" set JAASSOAP=-Djaassoap=off  @echo off  "%JAVA_EXE%"
%PERFJAVAOPTION% %WAS_LOGGING% %javaoption% %CONSOLE_ENCODING% %WAS_DEBUG% "%CLIENTSOAP%" "%JAASSOAP%" "%CLIENTSAS%"
"%CLIENTIPC%" "%CLIENTSSL%" %WSADMIN_PROPERTIES_PROP% %WORKSPACE_PROPERTIES% "-Duser.install.root=%USER_INSTALL_ROOT%"
"-Dwas.install.root=%WAS_HOME%" %wsadminTraceFile% %wsadminTraceString% %wsadminValOut% %wsadminHost% %wsadminConnType%
%wsadminPort% %wsadminLang% -classpath %C_PATH% com.ibm.ws.scripting.WasxShell %*  set RC=%ERRORLEVEL%  goto END  :END  @endlocal
 set MYERRORLEVEL=%ERRORLEVEL% if defined PROFILE_CONFIG_ACTION exit %MYERRORLEVEL% else exit /b %MYERRORLEVEL%  End of
wsadmin.bat
```

Auditing invocations of the wsadmin tool

Run the following wsadmin scripts as part of the environment setup: create the cluster definition, create data sources and JMS object configuration, or install one or more EAR files that comprise the hosted software on the application server. Each of the scripts, wsadmin and non- wsadmin, need to support the ability to capture a log of the activity performed when you run the script.

About this task

In order to set up your application server environment, you must perform multiple tasks. For example, the following non-wsadmin scripts: create the persistent session database, install the JDBC driver for the database on the system, set up MQ and create MQ queues on the system, or place PDF files in specific locations that are required as part of the application structure. You must also run the following wsadmin scripts as part of the environment setup: create the cluster definition, create data sources and JMS object configuration, or install one or more EAR files that comprise the hosted software on WebSphere Application Server. Each of the scripts, wsadmin and non- wsadmin, need to support the ability to capture a log of the activity performed when you run the script. All of the logs from the scripts are written in a specific directory that archives each time you create an environment. Each time you set up an environment, the overall process is considered a job and each job has an associated identifier. The identifier is a string that includes the date, environment name, machine name, operator, and approval code as indicated by company policy. To examine the logs at a later time, after the environment provisioning is complete, and verify that all of the log files for the wsadmin and non-wsadmin scripts reflect the actual output of the script that you ran for a specific job, and that no other logs are mixed in with the ones from that job, perform the following steps:

1. Start the wsadmin tool using the –jobid *string*, –appendTrace *string*, or the -tracefile *string* option. For more information about these options, see the “Wsadmin tool” on page 1181 article. For more information about starting the wsadmin tool, see the “Starting the wsadmin scripting client” on page 77 article. Use the -tracefile option to name the logs based on the activity performed by the script that you want to run and to locate the log files in the specific directory for the job. Uses the -appendtrade `true` option to append to an existing log file, if one already exists. Use the -jobid option to embed an identifier within the log file so that you can validate that all of the logs were the result of the same specific provisioning activity and not some other job.

You can change the name and location of a file. Modifying the contents of the log file can prove difficult. Also, different log files can have the same job ID and each log file needs a unique name. So the -jobid option provides an important audit and correlation function that the -tracefile option cannot provide.

2. Examine the log file for the job ID that you specified. Use the log files to audit or correlate the wsadmin tool.

Example

The following example outputs to the log of the wsadmin tool when you use th -jobid *string* parameter:

```
[5/16/05 15:45:49:449 CDT] 0000000a AbstractShell A   JobID= scriptTest1
```

Chapter 16. Troubleshooting with scripting

Use these topics to learn more about troubleshooting with scripting.

About this task

This topic contains the following tasks:

- "Tracing operations with the wsadmin tool"
- "Configuring traces using scripting" on page 1175
- "Turning traces on and off in servers processes using scripting" on page 1176
- "Dumping threads in server processes using scripting" on page 1177
- "Setting up profile scripts to make tracing easier using scripting" on page 1177
- "Enabling the Runtime Performance Advisor tool using scripting" on page 1178

Example

You can set the trace string using either of the following supported formats. For example:

```
$AdminControl trace com.ibm.*=all=enabled
```

or

```
$AdminControl trace com.ibm.*=all
```

For more information see the Tracing and logging configuration article and the Log level settings article in the *Troubleshooting and support* PDF.

Tracing operations with the wsadmin tool

You can enable tracing with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 77 article for more information.

About this task

Perform the following steps to trace operations:

Enable wsadmin client tracing with the following command:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=all
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=all')
  ```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`trace`	is an AdminControl command

com.ibm.*=all	indicates to turn on tracing

The following command disables tracing:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=info
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=info')
  ```

where:

$	is a Jacl operator for substituting a variable name with its value
AdminControl	is an object that enables the manipulation of MBeans running in a WebSphere server process
trace	is an AdminControl command
com.ibm.*=info	indicates to turn off tracing

The trace command changes the trace settings for the current session. You can change this setting persistently by editing the wsadmin.properties file. The property com.ibm.ws.scripting.traceString is read by the launcher during initialization. If it has a value, the value is used to set the trace.
A related property, com.ibm.ws.scripting.traceFile, designates a file to receive all trace and logging information. The wsadmin.properties file contains a value for this property. Run the wsadmin tool with a value set for this property. It is possible to run without this property set, where all logging and tracing goes to the administrative console.

Extracting properties files to troubleshoot your environment

Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

About this task

To debug problems in your environment, you can use the wsadmin tool to create a properties file to review your configuration. The properties file includes the most commonly used attributes or configuration data and values for the resource of interest. You can create a properties file for any of the following resources:

- Nodes
- Profiles
- Application servers
- Virtual hosts
- Authorization tables
- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Use properties files to troubleshoot your configuration. If you cannot resolve the error, you can provide IBM Support with a copy of the properties file.

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Extract the application server configuration of interest.

Use the extractConfigProperties command and the following command parameters to extract a specific object configuration

Parameter	Description
-propertiesFileName	Specifies the name of the properties file to extract. (String, required)
-configData	Specifies the configuration object instance in the format `Node=node1`. This parameter is required if you do not specify the configuration object name as the target object. (String, optional)
-options	Specifies additional configuration options, such as GENERATETEMPLATE=true. (Properties, optional)
-filterMechanism	Specifies filter information for extracting configuration properties. Specify All to extract all configuration properties. Specify NO_SUBTYPES to exclude properties specified with the selectedSubTypes parameter. Specify SELECTED_SUBTYPES to extract specific configuration properties specified with the selectedSubTypes parameter. (String, optional)
-selectedSubTypes	Specifies the configuration properties to include or exclude when the command extracts the properties. Specify this parameter if you set the filterMechanism parameter to NO_SUBTYPES or SELECTED_SUBTYPES. The following strings are examples of sever subtypes: ApplicationServer, EJBContainer. (String, optional)

The following example extracts the properties configuration for the `server1` application server:

```
AdminTask.extractConfigProperties('-propertiesFileName ConfigProperties_server1.props -configData Server=server1')
```

The system extracts the properties file, which contains each of the configuration objects and attributes for the `server1` application server.

You can also use the extractConfigProperties command to extract a specific object configuration from a deployment manager, as the following Jython example displays:

```
AdminTask.extractConfigProperties('-propertiesFileName ConfigProperties_server1.props -configData Server=dmgr')
```

The system extracts the properties file, which contains each of the configuration objects and attributes for the `dmgr` deployment manager.

Results

The system creates a properties file based on the resource configuration of interest.

Configuring traces using scripting

Use the wsadmin tool and scripting to configure traces for a configured server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to set the trace for a configured server:

1. Identify the server and assign it to the server variable:

- Using Jacl:

```
set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

- Using Jython:

```
server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print server
```

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

2. Identify the trace service belonging to the server and assign it to the tc variable:
   - Using Jacl:

```
set tc [$AdminConfig list TraceService $server]
```

   - Using Jython:

```
tc = AdminConfig.list('TraceService', server)
print tc
```

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
```

3. Set the trace string. The following example sets the trace string for a single component:
   - Using Jacl:

```
$AdminConfig modify $tc {{startupTraceSpecification
com.ibm.websphere.management.*=all=enabled}}
```

   - Using Jython:

```
AdminConfig.modify(tc, [['startupTraceSpecification',
'com.ibm.websphere.management.*=all=enabled']])
```

4. The following command sets the trace string for multiple components:
   - Using Jacl:

```
$AdminConfig modify $tc {{startupTraceSpecification
com.ibm.websphere.management.*=all=enabled:com.ibm.ws.
management.*=all=enabled:com.ibm.ws.runtime.*=all=enabled}}
```

   - Using Jython:

```
AdminConfig.modify(tc, [['startupTraceSpecification',
'com.ibm.websphere.management.*=all=enabled:com.ibm.ws.
management.*=all=enabled:com.ibm.ws.runtime.*=all=enabled']])
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 58 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 40 article for more information.

Turning traces on and off in servers processes using scripting

You can use scripting to turn traces on or off in server processes.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

Perform the following steps to turn traces on and off in server processes:

1. Identify the object name for the TraceService MBean running in the process:
   - Using Jacl:

```
$AdminControl completeObjectName type=TraceService,node=mynode,process=server1,*
```

- Using Jython:

  ```
  AdminControl.completeObjectName('type=TraceService,node=mynode,process=server1,*')
  ```

2. Obtain the name of the object and set it to a variable:
   - Using Jacl:

     ```
     set ts [$AdminControl completeObjectName type=TraceService,process=server1,*]
     ```

   - Using Jython:

     ```
     ts = AdminControl.completeObjectName('type=TraceService,process=server1,*')
     ```

3. Turn tracing on or off for the server. For example:
   - To turn tracing on, perform the following step:
     - Using Jacl:

       ```
       $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=enabled
       ```

     - Using Jython:

       ```
       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=enabled')
       ```

   - To turn tracing off, perform the following step:
     - Using Jacl:

       ```
       $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=disabled
       ```

     - Using Jython:

       ```
       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=disabled')
       ```

Dumping threads in server processes using scripting

Use the AdminControl object to dump the Java threads of a running server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

The following example produces a Java core file. You can use this file for problem determination.

- Using Jacl:

  ```
  set jvm [$AdminControl completeObjectName type=JVM,process=server1,*]
  $AdminControl invoke $jvm dumpThreads
  ```

- Using Jython:

  ```
  jvm = AdminControl.completeObjectName('type=JVM,process=server1,*')
  AdminControl.invoke(jvm, 'dumpThreads')
  ```

Setting up profile scripts to make tracing easier using scripting

You can use scripting and the wsadmin tool to set up profile scripts to facilitate tracing.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

Set up a profile script to make tracing easier. The following profile script example turns tracing on and off for `server1`:

- Using Jacl:

  ```
  proc ton {} {
    global AdminControl
    set ts [$AdminControl queryNames type=TraceService,node=mynode,process=server1,*]
    $AdminControl setAttribute $ts traceSpecification com.ibm.=all=enabled
  ```

```
}

proc toff {} {
  global AdminControl
  set ts [$AdminControl queryNames type=TraceService,node=mynode,process=server1,*]
  $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=disabled
}

proc dt {} {
  global AdminControl
  set jvm [$AdminControl queryNames type=JVM,node=mynode,process=server1,*]
  $AdminControl invoke $jvm dumpThreads
}
```

- Using Jython:

```
def ton():
       global lineSeparator
       ts = AdminControl.queryNames('type=TraceService,node=mynode,process=server1,*')

       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.=all=enabled')

def toff():
       global lineSeparator
       ts = AdminControl.queryNames('type=TraceService,node=mynode,process=server1,*')

       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=disabled')

def dt():
       global lineSeparator
       jvm = AdminControl.queryNames('type=JVM,node=mynode,process=server1,*')
       AdminControl.invoke(jvm, 'dumpThreads')
```

If you start the wsadmin tool with this profile script, you can use the **ton** command to turn on tracing in the server, the **toff** command to turn off tracing, and the **dt** command to dump the Java threads. For more information about running scripting commands in a profile script, see the “Starting the wsadmin scripting client” on page 77 article.

Enabling the Runtime Performance Advisor tool using scripting

You can configure the Runtime Performance Advisor using the wsadmin tool or the administrative console.

Before you begin

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 77 article for more information.

About this task

The Runtime Performance Advisor tool provides advice to help tune systems for optimal performance. See the Using the Runtime Performance Advisor article in the *Tuning guide* PDF for more information on how to enable this tool using the administrative console. The recommendations display as text in the SystemOut.log file.

The Runtime Performance Advisor (RPA) requires that the Performance Monitoring Service (PMI) is enabled. It does not require that individual counters be enabled. When a counter that is needed by the RPA is not enabled, the RPAr will enable it automatically.

There is no MBean/object available for wsadmin to create a RPA configuration. You can use wsadmin to change the settings and make them effective at runtime. These changes will not be persisted. The changes remain until you stop the server. Since the RPA is disabled once you stop the server, you may want to disable the PMI Service or the counters that were enabled while it was active. You can enable the following counters using the Runtime Performance Advisor:

```
ThreadPools (module)
Web Container (module)
Pool Size
Active Threads
Object Request Broker (module)
Pool Size
Active Threads
JDBC Connection Pools (module)
Pool Size
Percent used
Prepared Statement Discards
Servlet Session Manager (module)
External Read Size
External Write Size
External Read Time
External Write Time
No Room For New Session
System Data (module)
CPU Utilization
Free Memory
```

The following provides an explanation for some of the settings that you can use:

- Calculation interval PMI data - This setting is taken over an interval of time and averaged to provide advice. The calculation interval specifies the length of the time over which data is taken for this advice. Details within the advice messages will appear as averages over this interval.
- Maximum warning sequence - This setting refers to the number of consecutive warnings issued before the threshold is relaxed. For example, if the maximum warning sequence is set to 3, then the advisor only sends three warnings to indicate that the prepared statement cache is overflowing. After that, a new alert is only issued if the rate of discards exceeds the new threshold setting.
- Number of processors - This setting specifies the number of processors on the server. It is critical in order to ensure accurate advice for the specific configuration of the system.

To enable the Runtime Performance Advisor tool using the wsadmin tool, perform the following steps:

Setup the Runtime Performance Advisor (RPA), for example:

- Using Jacl:

```
set perf [$AdminControl queryNames mbeanIdentifier=ServerRuleDriverMBean2,process=server1,*]
set enabledVal [java::new java.lang.Boolean true]
set attr [java::new javax.management.Attribute enabled $enabledVal]
set perfObject [$AdminControl makeObjectName $perf]
set ObjectArray [java::new {java.lang.Object[]} 1]
set sigArray [java::new {java.lang.String[]} 1]
$ObjectArray set 0 $attr
$sigArray set 0 "javax.management.Attribute"
$AdminControl invoke_jmx $perfObject setRPAAttribute $ObjectArray  $sigArray

$AdminConfig save
```

What to do next

After completing the previous steps, start the server and monitor RPA.

AdministrationReports command group for the AdminTask object

You can use the Jython or Jacl scripting languages to troubleshoot your configuration with the wsadmin tool. The commands in the AdministrationReports group can be used to create a report of inconsistencies in your system configuration or a report that describes the port usage in the system.

The following commands are available for the AdministrationReports group of the AdminTask object:

- “reportConfigInconsistencies”
- “reportConfiguredPorts”

reportConfigInconsistencies

Use the **reportConfigInconsistencies** command to create a report of inconsistencies in the system configuration.

Target object

None

Required parameters and return values
- Parameters: None
- Returns: A report that describes inconsistencies found in the system.

Interactive mode example usage

Example output:

```
Configuration consistency report for  cell yardbirdCell  cells/yardbirdCell/test.xml is a zero
 length file. cells/yardbirdCell/nodes/DummyNode  does not contain a serverindex.xml  document.
 cells/yardbirdCell/applications/Test. ear/deployments/Test does not contain  a deployment.xml document.
 3 consistency problems were found.
```

reportConfiguredPorts

Use the **reportConfig uredPorts** command to create a report of all the ports configured in the cell.

Target object

None

Required parameters and return values
- Parameters: None
- Returns: A report that describes the port usage in the system.

Examples

Interactive mode example usage:

Example output:

```
Ports configured in cell yardbirdCell  Node yardbirdCellMgr / Server dmgr yardbird:7283
CELL_DISCOVERY_ADDRESS yardbird:9809 BOOTSTRAP_ADDRESS ...  Node dizzyNode1 / Server server1
dizzy:2813 BOOTSTRAP_ADDRESS dizzy:8880 SOAP_CONNECTOR_ADDRESS ...  Node dizzyNode1 / Server
nodeagent dizzy:2814 BOOTSTRAP_ADDRESS dizzy:9904 ORB_LISTENER_ADDRESS
```

Related tasks

“Using the AdminTask object for scripted administration” on page 58
Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Related reference

“Commands for the AdminTask object” on page 1310
Use the AdminTask object to run administrative commands with the wsadmin tool.

Chapter 17. Scripting and command line reference material

Use this topic to locate wsadmin tool commands for the AdminTask, AdminControl, AdminConfig, and AdminApp scripting objects. This topic also provides a pointer to command line commands and options.

About this task

All reference topics are located in the **Reference** section of the information center. Use the navigation paths described in this topic to locate specific reference information.

- View command line reference topics. This includes administrative commands such as the startServer, manageprofiles, and backupConfig commands. To view all command reference information, use the following navigation path in the information center:

  Reference > **Commands**
- View command reference topics for the wsadmin tool. This includes administrative scripting commands for the AdminTask, AdminConfig, AdminApp, and AdminControl objects. All commands are organized by command group name. To view all scripting reference information, use the following navigation path in the information center:

  Reference > **Administrator scripting interfaces**

Wsadmin tool

The wsadmin tool runs scripts. You can use the wsadmin tool to manage application server as well as the configuration, application deployment, and server runtime operations.

Note: All users who run commands from a specific profile must have authority to modify files that are created by other users that use the same profile. Otherwise, you might see a permission denied error in the log files. To avoid this issue, consider one of the following policies:

- Use specific profiles for distinct user authorities
- Always use the same user for all of the commands that are run in a given profile
- Ensure that all users of a specific profile belong to the same group. In addition, ensure that each user of a group has the read and write authority to the files that are created by other members in the same profile.

The options for the wsadmin tool are case insensitive. Do not pass in empty strings in place of command options. The wsadmin tool displays general help information if you specify an empty string as the command option. Use the following command-line invocation syntax for the wsadmin scripting client:

```
wsadmin [-h(help)]

[-?]

[-c <commands>]

[-p <properties_file_name>]

[-profile <profile_script_name>]

[-profileName <profile_name>]

[-f <script_file_name>]

[-javaoption java_option]

[-lang language]

[-wsadmin_classpath classpath]

[-conntype SOAP [-host host_name] [-port port_number] [-user user ID] [-password password]

[-conntype JSR160RMI [-host host_name] [-port port_number] [-user user ID] [-password password]
```

```
[-conntype RMI [-host host_name] [-port port_number] [-user user ID] [-password password]

[-conntype IPC [-ipchost host_name] [-port port_number] [-user user ID] [-password password]
[-jobid string]

[-tracefile trace_file]

[-appendtrace true/false]

[script parameters]
```

The element, `script parameters`, represents any argument other than the ones listed previously. The argc variable contains the number of arguments, and the argv variable contains a list of arguments in the order that they were coded.

Options

-c Specifies to run a single command. Multiple -c options can exist on the command line. They run in the order that you designate.

If you invoke the wsadmin tool with the -c option, any changes that you make to the configuration are saved automatically then. If you make configuration changes and you are not using the -c option, then you must use the **save** command of the AdminConfig object to save the changes. Read about saving configuration changes with the wsadmin tool for more information.

-f Specifies a script to run.

Only one -f option can exist on the command line.

You can use the -f option to run scripts that contain nested Jython scripts. In the following example, the test2 script imports the test1 script:

```
#test1.py

def   listServer():
       print AdminConfig.list("Server")

#test2.py

import test1
test1.listServer()
```

To run the caller script, run the following command from the install_root/bin directory:

```
wsadmin -lang jython -f  test2.py
```

After the scripts run, the system returns the following sample command output:

```
server1(cells/myCell/nodes/myNode/servers/myServer|server.xml#Server_1183122130078)
```

-javaoption
Specifies a valid Java standard or a non-standard option. Multiple -javaoption options can exist on the command line.

To shorten the length of the command, type the command in the following way:

```
wsadmin -javaoption java_option  java_option
```

instead of the:

```
wsadmin -javaoption java_option -javaoption java_option
```

-lang
Specifies the language of the script file, the command, or an interactive shell. The possible languages include: Jacl and Jython. The options for the -lang argument include: jacl and jython.

This option overrides language determinations that are based on a script file name, a profile script file name, or the com.ibm.ws.scripting.defaultLang property. The -lang argument has no default value.

If you do not specify the -lang argument but you have the -f *<script_file_name>* argument specified, then the wsadmin tool determines the language based on a target script file name. If you do not specify the -lang argument and the -f argument, the wsadmin tool determines the language based on

a profile script file name if the -profile <profile_script_name> argument is specified. If the command line or the property does not supply the script language, and the wsadmin tool cannot determine it, then an error message is generated.

-p

Specifies a properties file.

The file listed after -p, represents a Java properties file that the scripting process reads. Three levels of default properties files load before the properties file that you specify on the command line. The first level is the installation default, wsadmin.properties, which is located in the product properties directory. The second level is the user default, wsadmin.properties, which is located in your home directory. The third level is the properties file to which the environment variable `WSADMIN_PROPERTIES` references.

Multiple -p options can exist on the command line. Those options invoke in the order that you supply them.

-profile

Specifies a profile script.

The profile script runs before other commands, or scripts. If you specify `-c`, then the profile script runs before it invokes this command. If you specify `-f`, then the profile script runs before it runs the script. In interactive mode, you can use the profile script to perform any standard initialization that you want. You can specify multiple -profile options on the command line, and they invoke in the order that you supply them.

-profileName

Specifies the profile from which the wsadmin tool runs. Specify this option if one the following reasons apply:

- You run the wsadmin tool from the WAS_HOME/bin directory, and you do not have a default profile, or you want to run in a profile other than the default profile.
- You are currently in a profile bin directory but want to run the wsadmin tool from a different profile.

Note: WebSphere Application Server running on z/OS platforms does not support user-created profiles; only the default profile is used.

-?

Provides syntax help.

-help

Provides syntax help.

-conntype

Specifies the type of connection to use.

This argument consists of a string that determines the type, for example, SOAP, and the options that are specific to that connection type. Possible types include: `SOAP`, `RMI`, `JSR160RMI`, `IPC` and `NONE`. For each connection type, you can specify additional attributes about the connection.

For the `SOAP` connection type, you can specify the following attributes:

Attribute	Description
[-host host_name]	Specifies the host name for the connection.
[-port port_number]	Specifies the port number for the connection.
[-user userid]	Specifies the user ID to use to establish the connection.
[-password password]	Specifies the password to use to establish the connection.

For the `RMI` connection type, you can specify the following attributes:

Attribute	Description
[-host host_name]	Specifies the host name for the connection.

Attribute	Description
[-port port_number]	Specifies the port number for the connection.
[-user userid]	Specifies the user ID to use to establish the connection.
[-password password]	Specifies the password to use to establish the connection.

For the `JSR160RMI` connection type, you can specify the following attributes:

Attribute	Description
[-host host_name]	Specifies the host name for the connection.
[-port port_number]	Specifies the port number for the connection.
[-user userid]	Specifies the user ID to use to establish the connection.
[-password password]	Specifies the password to use to establish the connection.

For the `IPC` connection type, you can specify the following attributes:

Attribute	Description
[-ipchost host_name]	Specifies the host name for the connection. This attribute overrides the host name specified for the com.ibm.ws.scripting.ipchost property in the wsadmin.properties properties file.
[-port port_number]	Specifies the port number for the connection.
[-user userid]	Specifies the user ID to use to establish the connection.
[-password password]	Specifies the password to use to establish the connection.

Use the `-conntype NONE` option to run in local mode. The result is that the scripting client is not connected to a running server. You can manage server configuration, the installation and the uninstallation of applications without the application server running.

Note: You should eventually switch from the Remote Method Invocation (RMI) connector to the JSR160RMI connector because support for the RMI connector is deprecated.

-wsadmin_classpath

Use this option to make additional classes available to your scripting process.

Use the following option with a class path string:

```
/home/MyDir/Myjar.jar;/yourdir/yourdir.jar
```

The class path is then added to the class loader for the scripting process.

You can also specify this option in a properties file that is used by the wsadmin tool. The property is com.ibm.ws.scripting.classpath. If you specify -wsadmin_classpath on the command line, the value of this property overrides any value that is specified in a properties file. The class path property and the command-line options are not concatenated.

-host

Specify a host name to which wsadmin attempts to connect. The default wsadmin.properties file located in the properties directory of each profile provides `localhost` as the value of the host property, if this option is not specified.

-password

Specify a password to be used by the connector to connect to the server, if security is enabled in the server.

Note: On UNIX systems, the use of -password option might result in security exposure as the password information becomes visible to the system status program such as **ps** command which can be invoked by another user to display all the running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

-user or -username
Specifies a user name to be used by the connector to connect to the server if security is enabled in the server.

-port
Specifies a port to be used by the connector. The default wsadmin.properties file located in the properties directory of each application server profile provides a value in the port property to connect to the local server.

-jobid
Specifies a jobID string so that you can keep track of each invocation of the wsadmin tool for auditing purposes. The jobID string (jobID=xxxx) is displayed at the beginning of the wsadmin log file.

-tracefile
Specifies the name of the log file and location where the log output is directed. This option overrides the com.ibm.ws.scripting.traceFile property in the wsadmin.properties file.

-appendtrace
Determines if a trace appends to or overrides the end of the existing log file. Specify `true` to append the trace to the end of a log file or specify `false` to override the log file for each wsadmin invocation. The default value is `false`.

The following example specifies the jobID option, log location and appends the trace to the log file.

```
wsadmin -jobid wsadmin_test_1 -tracefile /temp/wsadmin_test_1.log -appendtrace true
```

In the following syntax examples, *mymachine* is the name of the host in the wsadmin.properties file that is specified by the com.ibm.ws.scripting.port property:

SOAP connection to the local host
Use the options that are defined in the `wsadmin.properties` file.

SOAP connection to the *mymachine* host
Using Jacl, enter the following example code:

```
wsadmin -f test1.jacl -profile setup.jacl -conntype SOAP
-port mymachinesoapportnumber -host mymachine
```

Using Jython:

```
wsadmin -lang jython -f  test1.py -profile setup.py -conntype
SOAP -port mymachinesoapportnumber -host mymachine
```

Initial and maximum Java heap size
Using Jacl:

```
wsadmin -javaoption -Xms128m -Xmx256m -f test.jacl
```

Using Jython:

```
wsadmin -lang jython -javaoption -Xms128m -Xmx256m -f test.py
```

JSR160RMI connection with security
Using Jacl:

```
wsadmin -conntype JSR160RMI -port  JSR160rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin -lang jython -conntype JSR160RMI -port  JSR160portnumber -user userid
-password password
```

The element, *rmiportnumber*, for your connection is displayed in the administrative console as BOOTSTRAP_ADDRESS.

Note: On UNIX systems, the use of -password option might result in security exposure as the password information becomes visible to the system status program such as **ps** command which can be invoked by another user to display all the running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

RMI connection with security

Using Jacl:

```
wsadmin -conntype RMI -port  rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin -lang jython -conntype RMI -port  rmiportnumber -user userid
-password password
```

The element, *rmiportnumber*, for your connection is displayed in the administrative console as BOOTSTRAP_ADDRESS.

Note: On UNIX systems, the use of -password option might result in security exposure as the password information becomes visible to the system status program such as **ps** command which can be invoked by another user to display all the running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

Local mode of operation to perform a single command

Using Jacl:

```
wsadmin -conntype NONE -c "$AdminApp uninstall app"
```

Using Jython:

```
wsadmin -lang jython -conntype NONE -c "AdminApp.uninstall('app')"
```

wsadmin tool performance tips

Follow these tips to get the best performance from the wsadmin tool.

The following performance tips are for the wsadmin tool:

- If the deployment manager is running at a higher service maintenance level than that of the node agent, you must run the `wsadmin.sh` or the `wsadmin.bat` from the `bin` directory of the deployment manager.
- When you launch a script using the wsadmin tool, a new process is created with a new Java virtual machine (JVM) API. If you use scripting with multiple wsadmin **-c** commands from a batch file or a shell

script, these commands run slower than if you use a single wsadmin **-f** command. The -f option runs faster because only one process and JVM API are created for installation and the Java classes for the installation load only once.

The following example, illustrates running multiple application installation commands from a batch file.

Using Jacl:

```
wsadmin -c "$AdminApp install /home/myDir/myApps/App1.ear {-appname appl1}"
wsadmin -c "$AdminApp install /home/myDir/myApps/App2.ear {-appname appl2}"
wsadmin -c "$AdminApp install /home/myDir/myApps/App3.ear {-appname appl3}"
```

Using Jython:

```
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App1.ear', '[-appname appl1]')"
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App2.ear', '[-appname appl2]')"
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App3.ear', '[-appname appl3]')"
```

Or, for example, using Jacl, you can create the *appinst.jacl* file that contains the commands:

```
$AdminApp install /home/myDir/myApps/App1.ear {-appname appl1}
$AdminApp install /home/myDir/myApps/App2.ear {-appname appl2}
$AdminApp install /home/myDir/myApps/App3.ear {-appname appl3}
```

Invoke this file using the following command: `wsadmin -f appinst.jacl`

Or using Jython, you can create the *appinst.py* file, that contains the commands:

```
AdminApp.install('/home/myDir/myApps/App1.ear', '[-appname appl1]')
AdminApp.install('/home/myDir/myApps/App2.ear', '[-appname appl2]')
AdminApp.install('/home/myDir/myApps/App3.ear', '[-appname appl3]')
```

Then invoke this file using the following command: `wsadmin -lang jython -f appinst.py`.

- Use the AdminControl **queryNames** and **completeObjectName** commands carefully with a large installation. For example, if only a few beans exist on a single machine, the `$AdminControl queryNames *` command performs well.

  If a scripting client connects to the deployment manager in a multiple machine environment, use a command only if it is necessary for the script to obtain a list of all the MBeans in the system. If you need the MBeans on a node, it is easier to invoke `"$AdminControl queryNames node=mynode,*"`. The JMX system management infrastructure forwards requests to the system to fulfill the first query, `*`. The second query, `node=mynode,*` is targeted to a specific machine.
- The WebSphere Application Server is a distributed system, and scripts perform better if you minimize remote requests. If some action or interrogation is required on several items, for example, servers, it is more efficient to obtain the list of items once and iterate locally. This procedure applies to the actions that the AdminControl object performs on running MBeans, and actions that the AdminConfig object performs on configuration objects.

Commands for the Help object

You can use the Jython or Jacl scripting languages to find general help and dynamic online information about the currently running MBeans with the wsadmin tool. Use the Help object as an aid in writing and running scripts with the AdminControl object.

The following commands are available for the Help object:

AdminApp

Use the **AdminApp** command to view a summary of each available method for the AdminApp object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7095I: The AdminApp object allows application objects to
be manipulated -- this includes installing, uninstalling, editing,
and listing.  Most of the commands supported by AdminApp operate in two
modes: the default mode is one in which AdminApp communicates with the
WebSphere Application Server to accomplish its tasks.  A local mode is also
possible, in which no server communication takes place.  The local
mode of operation is invoked by bringing up the scripting client with
no server connected using the command line "-conntype NONE" option or setting the
"com.ibm.ws.scripting.connectionType=NONE" property in the wsadmin.properties.

The following commands are supported by AdminApp; more detailed information
about each of these commands is available by using the "help" command of AdminApp and
supplying the name of the command as an argument.

deleteUserAndGroupEntries  Deletes all the user/group information for all the roles and all
the username/password information for RunAs roles for a given application.

edit Edit the properties of an application

editInteractive Edit the properties of an application interactively

export Export application to a file

exportDDL Export DDL from application to a directory

help Show help information

install Installs an application, given a file name and an option string.

installInteractive Installs an application in interactive mode, given a file name
and an option string.

isAppReady Checks whether the application is ready to be run

list List all installed applications, either all applications or applications on a given target scope.

listModules List the modules in a specified application

options Shows the options available, either for a given file, or in general.
```

```
publishWSDL Publish WSDL files for a given application

taskInfo Shows detailed information pertaining to a given installation task for a given file

uninstall Uninstalls an application, given an application name and an option string

updateAccessIDs Updates the user/group binding information with accessID from user
registry for a given application

view View an application or module, given an application or module name
```

Examples

- Using Jacl:

  ```
  $Help AdminApp
  ```

- Using Jython:

  ```
  print Help.AdminApp()
  ```

AdminConfig

Use the **AdminConfig** command to view a summary of each available method for the AdminConfig object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7053I: The following functions are
supported by AdminConfig:


create Creates a configuration object,
given a type, a parent, and

a list of attributes


create Creates a configuration object,
given a type, a parent, a

list of attributes, and an attribute
name for the new object

remove Removes the specified configuration
object

list Lists all configuration objects of
a given type

list Lists all configuration objects of
a given type, contained

within the scope supplied
```

```
show Show all the attributes of a given
configuration object

show Show specified attributes of a
given configuration object

modify Change specified attributes of a
given configuration object

getId Show the configId of an object,
given a string version of

its containment

contents Show the objects which a given
type contains

parents Show the objects which contain
a given type

attributes Show the attributes for a
given type

types Show the possible types for
configuration

help Show help information
```

Examples

- Using Jacl:

  ```
  $Help AdminConfig
  ```

- Using Jython:

  ```
  print Help.AdminConfig()
  ```

AdminControl

Use the **AdminControl** command to view a summary of the help commands and ways to invoke an administrative command.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7027I: The following functions are supported by AdminControl:

getHost returns String representation of connected host

getPort returns String representation of port in use

getType returns String representation of connection type in use

reconnect reconnects with server
```

```
queryNames Given ObjectName and QueryExp, retrieves set of ObjectNames that match.

queryNames Given String version of ObjectName, retrieves String of ObjectNames that match.

getMBeanCount returns number of registered beans

getDomainName returns "WebSphere"

getDefaultDomain returns "WebSphere"

getMBeanInfo Given ObjectName, returns MBeanInfo structure for MBean

isInstanceOf Given ObjectName and class name, true if MBean is of that class

isRegistered true if supplied ObjectName is registered

isRegistered true if supplied String version of ObjectName is registered

getAttribute Given ObjectName and name of attribute, returns value of attribute

getAttribute Given String version of ObjectName and name of attribute, returns value of attribute

getAttributes Given ObjectName and array of attribute names, returns AttributeList

getAttributes Given String version of ObjectName and attribute names,
returns String of name value pairs

setAttribute Given ObjectName and Attribute object, set attribute for MBean specified

setAttribute Given String version of ObjectName, attribute name and attribute value,
set attribute for MBean specified

setAttributes Given ObjectName and AttributeList object, set attributes for the MBean specified

invoke Given ObjectName, name of method, array of parameters and signature, invoke method
on MBean specified

invoke Given String version of ObjectName, name of method, String version of parameter list, invoke
method on MBean specified.

invoke Given String version of ObjectName, name of method, String version of parameter list, and
String version of array of signatures, invoke method on MBean specified.

makeObjectName Return an ObjectName built with the given string

completeObjectName Return a String version of an object name given a template name

trace Set the wsadmin trace specification

help Show help information
```

Examples

- Using Jacl:

  ```
  $Help AdminControl
  ```

- Using Jython:

  ```
  print Help.AdminControl()
  ```

AdminTask

Use the **AdminTask** command to view a summary of help commands and ways to invoke an administrative command with the AdminTask object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX8001I: The AdminTask object enables the available administrative commands. AdminTask commands
operate in two modes: the default mode is one whichAdminTask communicates with the WebSphere Application
Server to accomplish its task. A local mode is also available in which no server communication takes place.
The local mode of operation is invoked by bringing up the scripting client using the command line
"-conntype NONE" option or setting the"com.ibm.ws.scripting.connectiontype=NONE" property in
wsadmin.properties file.

The number of administrative commands varies and depends on your WebSphere Application Server
installation. Use the following help commands to obtain a list of supported commands and their parameters:

help -commands        list all the administrative commands
help -commandGroups     list all the administrative command groups
help commandName       display detailed information for the specified command
help commandName stepName  display detailed information for the specified step belonging to the specified command
help commandGroupName    display detailed information for the specified command group

There are various flavors to invoke an administrative  command. They are

commandName invokes an administrative command that does not require any argument.

commandName targetObject invokes an admin command with the target object string, for example, the configuration
object name of a resource adapter. The expected target object varies with the administrative command invoked.

Use help command to get information on the target object of an administrative command.

commandName  options invokes an administrative command with the specified option strings. This invocation syntax
is used to invoke an administrative command that does not require a target object.

It is also used to enter interactive mode if "-interactive" mode is included in the options string.

commandName targetObject options invokes an administrative command with the specified target object and options strings.

If "-interactive" is included in the options string, then interactive mode is entered.
The target object and options strings vary depending on the admin command invoked.
Use help command to get information on the target object and options.
```

Examples

- Using Jacl:

  ```
  $AdminTask help
  ```

- Using Jython:

  ```
  print AdminTask.help()
  ```

all

Use the **all** command to view a summary of the information that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Name: WebSphere:cell=pongo,name=TraceService,mbeanIdentifier=cells/pongo/nodes/pongo/servers/server1/
server.xml#TraceService_1,type=TraceService,node=pongo,process=server1
Description: null
Class name: javax.management.modelmbean.RequiredModelMBean

Attribute                       Type                            Access
ringBufferSize                  int                             RW
traceSpecification              java.lang.String                RW

Operation
int getRingBufferSize()
void setRingBufferSize(int)
java.lang.String getTraceSpecification()
void setTraceState(java.lang.String)
void appendTraceString(java.lang.String)
void dumpRingBuffer(java.lang.String)
void clearRingBuffer()
[Ljava.lang.String; listAllRegisteredComponents()
[Ljava.lang.String; listAllRegisteredGroups()
[Ljava.lang.String; listComponentsInGroup
(java.lang.String)
[Lcom.ibm.websphere.ras.TraceElementState;
getTracedComponents()
[Lcom.ibm.websphere.ras.TraceElementState;
getTracedGroups()
java.lang.String getTraceSpecification(java.
lang.String)
void processDumpString(java.lang.String)
void checkTraceString(java.lang.String)
void setTraceOutputToFile(java.lang.String,
int, int, java.lang.String)
void setTraceOutputToRingBuffer(int, java.
lang.String)
java.lang.String rolloverLogFileImmediate
(java.lang.String, java.lang.String)


Notifications
jmx.attribute.changed

Constructors
```

Examples

- Using Jacl:

  ```
  $Help all [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.all(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

attributes

Use the **attributes** command to view a summary of all the attributes that the MBean defines by name. If you provide the MBean name parameter, the command displays information about the attributes,

operations, constructors, description, notifications, and classname of the specified MBean. If you specify the MBean name and attribute name, the command displays information about the specified attribute for the specified MBean.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

attribute name
Specifies the attribute of interest. (String)

Sample output

```
Attribute Type Access

ringBufferSize java.lang.Integer RW

traceSpecification string RW
```

Examples

- Using Jacl:
  ```
  $Help attributes [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```
- Using Jython:
  ```
  print Help.attributes(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

classname

Use the **classname** command to view a class name that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
javax.management.modelmbean.RequiredModelMBean
```

Examples

- Using Jacl:
  ```
  $Help classname [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```
- Using Jython:
  ```
  print Help.classname(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

constructors

Use the **constructors** command to view a summary of all of the constructors that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Constructors
```

Examples

- Using Jacl:

  ```
  $Help constructors [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.constructors(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

description

Use the **description** command to view a description that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Managed object for overall server process.
```

Examples

- Using Jacl:

  ```
  $Help description [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.description(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

help

Use the **help** command to view a summary of all the available methods for the Help object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7028I: The Help object has two purposes:

First, provide general help information for the objects supplied by the wsadmin tool
for scripting: Help, AdminApp, AdminConfig, and AdminControl.

Second, provide a means to obtain interface information about the MBeans that run in
the system.  For this purpose, a variety of commands are available to get
information about the operations, attributes, and other interface information
about particular MBeans.

The following commands are supported by Help; more detailed information about
each of these commands is available by using the "help" command of Help and by
supplying the name of the command as an argument.

attributes     given an MBean, returns help for attributes
operations     given an MBean, returns help for operations
constructors   given an MBean, returns help for constructors
description    given an MBean, returns help for description
notifications  given an MBean, returns help for notifications
classname      given an MBean, returns help for class name
all            given an MBean, returns help for all the previous
help           returns this help text
AdminControl    returns general help text for the AdminControl object
AdminConfig    returns general help text for the AdminConfig object
AdminApp       returns general help text for the AdminApp object
AdminTask      returns general help text for the AdminTask object
wsadmin        returns general help text for the wsadmin script launcher
message        given a message ID, returns an explanation and a user action
```

Examples

- Using Jacl:

  ```
  $Help help
  ```

- Using Jython:

  ```
  print Help.help()
  ```

message

Use the **message** command to view information for a message ID.

Target object

None.

Required parameters

message ID
Specifies the message ID of the message of interest. (String)

Optional parameters

None.

Sample output

```
Explanation: The container was unable to
passivate an enterprise bean due to
exception {2}
User action: Take action based upon
message in exception {2}
```

Examples

- Using Jacl:

  ```
  $Help message CNTR0005W
  ```

- Using Jython:

  ```
  print Help.message('CNTR0005W')
  ```

notifications

Use the **notifications** command to view a summary of all the notifications that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name of the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Notification

websphere.messageEvent.audit

websphere.messageEvent.fatal

websphere.messageEvent.error

websphere.seriousEvent.info

websphere.messageEvent.warning

jmx.attribute.changed
```

Examples

- Using Jacl:

  ```
  $Help notifications [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

```
print Help.notifications(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
```

operations

Use the **operations** command with the MBean name parameter to view a summary of all the operations that the MBean defines by name. Specify a value for the MBean name and operation name to display the signature of the operation for the MBean that is defined by name.

Target object

None.

Required parameters

MBean name
Specifies the object name of the MBean of interest. (String)

Optional parameters

operation name
Specifies the operation of interest. (String)

Sample output

The command returns output that is similar to the following example if you specify only the MBean name parameter:

```
Operation
int getRingBufferSize()
void setRingBufferSize(int)
java.lang.String getTraceSpecification()
void setTraceState(java.lang.String)
void appendTraceString(java.lang.String)
void dumpRingBuffer(java.lang.String)
void clearRingBuffer()
[Ljava.lang.String; listAllRegisteredComponents()
[Ljava.lang.String; listAllRegisteredGroups()
[Ljava.lang.String; listComponentsInGroup(java.lang.String)
[Lcom.ibm.websphere.ras.TraceElementState; getTracedComponents()
[Lcom.ibm.websphere.ras.TraceElementState; getTracedGroups()
java.lang.String getTraceSpecification(java.lang.String)
void processDumpString(java.lang.String)
void checkTraceString(java.lang.String)
void setTraceOutputToFile(java.lang.String, int, int, java.lang.String)
void setTraceOutputToRingBuffer(int, java.lang.String)
java.lang.String rolloverLogFileImmediate(java.lang.String, java.lang.String)
```

The command returns output that is similar to the following example if you specify the MBean name and operation name parameters:

```
void processDumpString(string)

Description: Write the contents of the Ras
services ring buffer to the specified file.

Parameters:

Type         string
Name         dumpString
Description  A String in the specified format to process or null.
```

Examples

- Using Jacl:

```
$Help operations [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
$Help operations [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*] processDumpString
```

- Using Jython:

```
print Help.operations(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
print Help.operations(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'), 'processDumpString')
```

Related concepts

“Help object for scripted administration” on page 27
The Help object provides general help, online information about running MBeans, and help on messages.

Related tasks

Chapter 17, “Scripting and command line reference material,” on page 1181
Use this topic to locate wsadmin tool commands for the AdminTask, AdminControl, AdminConfig, and AdminApp scripting objects. This topic also provides a pointer to command line commands and options.

Commands for the AdminConfig object

Use the AdminConfig object to invoke configuration commands and to create or change elements of the WebSphere Application Server configuration, for example, creating a data source.

You can start the scripting client without a running server, if you only want to use local operations. To run in local mode, use the -conntype NONE option to start the scripting client. You receive a message that you are running in the local mode. If a server is currently running, running the AdminConfig tool in local mode is not recommended. This is because any configuration changes made in local mode will not be reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager.

When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

The following commands are available for the AdminConfig object:

attributes

Use the **attributes** command to return a list of the top level attributes for a given type.

Target object

None.

Required parameters

object type
Specifies the name of the object type that is based on the XML configuration files. The object type does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

```
"properties Property*" "serverSecurity ServerSecurity"
"server Server@" "id Long" "stateManagement StateManageable"
"name String" "moduleVisibility EEnumLiteral(MODULE,
COMPATIBILITY, SERVER, APPLICATION)" "services Service*"
"statisticsProvider StatisticsProvider"
```

Examples

- Using Jacl:

  ```
  $AdminConfig attributes ApplicationServer
  ```

- Using Jython:

  ```
  print AdminConfig.attributes('ApplicationServer')
  ```

checkin

Use the **checkin** command to check a file into the configuration repository that is described by the document Uniform Resource Identifier (URI). This method only applies to deployment manager configurations.

Target object

None.

Required parameters

URI
The document URI is relative to the root of the configuration repository, for example:

.

file name
Specifies the name of the source file to check in.

opaque object
Specifies an object that the **extract** command of the AdminConfig object returns by a prior call.

Optional parameters

None.

Sample output

```
"properties Property*" "serverSecurity ServerSecurity"
"server Server@" "id Long" "stateManagement StateManageable"
"name String" "moduleVisibility EEnumLiteral(MODULE,
COMPATIBILITY, SERVER, APPLICATION)" "services Service*"
"statisticsProvider StatisticsProvider"
```

Examples

- Using Jacl:
- Using Jython:

convertToCluster

Use the **convertToCluster** command to convert a server so that it is the first member of a new server cluster.

Target object

None.

Required parameters

server ID
The configuration ID of the server of interest.

cluster name
Specifies the name of the cluster of interest.

Optional parameters

None.

Sample output

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set serverid [$AdminConfig getid /Server:myServer/]
$AdminConfig convertToCluster $serverid myCluster
```

- Using Jython:

```
serverid = AdminConfig.getid('/Server:myServer/')
print AdminConfig.convertToCluster(serverid, 'myCluster')
```

create

Use the **create** command to create configuration objects.

Target object

None.

Required parameters

type
Specifies the name of the object type that is based on the XML configuration files. This parameter value does not have to be the same name that the administrative console displays.

parent ID
Specifies the configuration ID of the parent object.

attributes
Specifies any attributes to add to the configuration ID.

Optional parameters

None.

Sample output

This command returns a string of the configuration object name, as this sample output displays:

```
ds1(cells/mycell/nodes/DefaultNode/servers/server1|resources.xml#DataSource_6)
```

Examples

- Using Jacl:

```
set jdbc1 [$AdminConfig getid /JDBCProvider:jdbc1/]
$AdminConfig create DataSource $jdbc1 {{name ds1}}
```

- Using Jython string attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')
print AdminConfig.create('DataSource', jdbc1, '[[name ds1]]')
```

- Using Jython with object attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')
print AdminConfig.create('DataSource', jdbc1, [['name', 'ds1']])
```

createClusterMember

Use the **createClusterMember** command to create a new server object on the node that the node id parameter specifies. This server is created as a new member of the existing cluster that is specified by the cluster id parameter, and contains attributes that are specified in the member attributes parameter. The server is created using the server template that is specified by the template id attribute, and that contains the name specified by the memberName attribute. The memberName attribute is required. The template options are available only for the first cluster member that you create. All cluster members that you create after the first member will be identical.

Target object

None.

Required parameters

cluster ID
Specifies the configuration ID of the cluster of interest.

node ID
Specifies the configuration ID of the node of interest.

template ID
Specifies the template ID to use to create the server.

member attributes
Specifies any attributes to add to the cluster member. The memberName attribute is required, and defines the name of the cluster member to create.

Optional parameters

None.

Sample output

This command returns the configuration ID of the newly created cluster member, as the following example displays:

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set clid [$AdminConfig getid /ServerCluster:myCluster/]
set nodeid [$AdminConfig getid /Node:mynode/]
$AdminConfig createClusterMember $clid $nodeid {{memberName newMem1} {weight 5}}
```

- Using Jython string attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
print AdminConfig.createClusterMember(clid, nodeid, '[[memberName newMem1] [weight 5]]')
```

- Using Jython with object attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
print AdminConfig.createClusterMember(clid, nodeid, [['memberName', 'newMem1'], ['weight', 5]])
```

createDocument

Use the **createDocument** command to create a new document in the configuration repository.

Target object

None.

Required parameters

document URI
Specifies the name of the document to create in the repository.

file name
Specifies a valid local file name of the document to create.

Optional parameters

None.

Examples

- Using Jacl:
- Using Jython with string attributes:

createUsingTemplate

Use the **createUsingTemplate** command to create a type of object with the given parent, using a template. You can only use this command for creation of a server with `APPLICATION_SERVER` type. If you want to create a server with a type other than `APPLICATION_SERVER`, use the **createGenericServer** or the **createWebServer** command.

Target object

None.

Required parameters

type
Specifies the type of object to create.

parent
Specifies the configuration ID of the parent.

template
Specifies a configuration ID of an existing object. This object can be a template object returned by using the listTemplates command, or any other existing object of the correct type.

Optional parameters

attributes
Specifies attribute values for the object. The attributes specified using this parameter override the settings in the template.

Sample output

The command returns the configuration ID of the new object, as the following example displays:

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set node [$AdminConfig getid /Node:mynode/]
set templ [$AdminConfig listTemplates JDBCProvider "DB2 JDBC Provider (XA)"]
$AdminConfig createUsingTemplate JDBCProvider $node {{name newdriver}} $templ
```

- Using Jython with string attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)")
print AdminConfig.createUsingTemplate('JDBCProvider', node, '[[name newdriver]]', templ)
```

- Using Jython with object attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)")
print AdminConfig.createUsingTemplate('JDBCProvider', node, [['name', 'newdriver']], templ)
```

defaults

Use the **defaults** command to display the default values for attributes of a given type. This method displays all of the possible attributes contained by an object of a specific type. If the attribute has a default value, this method also displays the type and default value for each attribute.

Target object

None.

Required parameters

type
Specifies the type of object to return. The name of the object type that you specify is based on the XML configuration files. This name does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The command returns string that contains a list of attributes with its type and value, as the following example displays:

```
Attribute               Type     Default

usingMultiRowSchema     Boolean  false
maxInMemorySessionCount Integer  1000
allowOverflow           Boolean  true
scheduleInvalidation    Boolean  false
writeFrequency          ENUM
writeInterval           Integer  120
writeContents           ENUM
invalidationTimeout     Integer  30
invalidationSchedule    InvalidationSchedule
```

Examples

- Using Jacl:

```
$AdminConfig defaults TuningParams
```

- Using Jython:

```
print AdminConfig.defaults('TuningParams')
```

deleteDocument

Use the **deleteDocument** command to delete a document from the configuration repository.

Target object

None.

Required parameters

documentURI
Specifies the document to delete from the repository.

Optional parameters

None.

Examples

- Using Jacl:
  ```
  $AdminConfig deleteDocument cells/mycell/myfile.xml
  ```
- Using Jython:
  ```
  AdminConfig.deleteDocument('cells/mycell/myfile.xml')
  ```

existsDocument

Use the **existsDocument** command to test for the existence of a document in the configuration repository.

Target object

None.

Required parameters

documentURI
Specifies the document to test for in the repository.

Optional parameters

None.

Sample output

The command returns a true value if the document exists, as the following example displays:

```
1
```

Examples

- Using Jacl:
  ```
  $AdminConfig existsDocument cells/mycell/myfile.xml
  ```
- Using Jython:
  ```
  print AdminConfig.existsDocument('cells/mycell/myfile.xml')
  ```

extract

Use the **extract** command to extract a configuration repository file that is described by the document URI and places it in the file named by `filename`. This method only applies to deployment manager configurations.

Target object

None.

Required parameters

documentURI
Specifies the document to extract from the configuration repository. The document URI must exist in the repository. The document URI is relative to the root of the configuration repository, for example:

filename
Specifies the filename to extract the document to. The filename must be a valid local filename where the contents of the document are written. If the file that is specified by the filename parameter exists, the extracted file replaces it.

Optional parameters

None.

Sample output

The command returns an opaque ″digest″ object which should be used to check the file back in using the **checkin** command.

Examples

- Using Jacl:
- Using Jython:

getCrossDocumentValidationEnabled

Use the **getCrossDocumentValidationEnabled** command to return a message with the current cross-document enablement setting. This method returns true if cross-document validation is enabled.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns string that contains the message with the cross-document validation setting, as the following example displays:

```
WASX7188I: Cross-document validation enablement set to true
```

Examples

- Using Jacl:

```
$AdminConfig getCrossDocumentValidationEnabled
```

- Using Jython:

```
print AdminConfig.getCrossDocumentValidationEnabled()
```

getid

Use the **getid** command to return the configuration ID of an object.

Target object

None.

Required parameters

containment path
Specifies the containment path of interest.

Optional parameters

None.

Sample output

The command returns configuration ID for an object that is described by the containment path, as the following example displays:

```
Db2JdbcDriver(cells/testcell/nodes/testnode|resources.xml#JDBCProvider_1)
```

Examples

- Using Jacl:

```
$AdminConfig getid /Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/
```

- Using Jython:

```
print AdminConfig.getid('/Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/')
```

getObjectName

Use the **getObjectName** command to return a string version of the object name for the corresponding running MBean. This method returns an empty string if no corresponding running MBean exists.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object name to return.

Optional parameters

None.

Sample output

The command returns a string that contains the object name, as the following example displays:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=cells/mycell/nodes/mynode/servers/server1/
server.xml#Server_1,type=Server,node=mynode,process=server1,processType=UnManagedProcess
```

Examples

- Using Jacl:

```
set server [$AdminConfig getid /Node:mynode/Server:server1/]
$AdminConfig getObjectName $server
```

- Using Jython:

```
server = AdminConfig.getid('/Node:mynode/Server:server1/')
print AdminConfig.getObjectName(server)
```

getObjectType

Use the **getObjectType** command to display the object type for the object configuration ID of interest.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object name to return.

Optional parameters

None.

Examples

- Using Jacl:

```
set server [$AdminConfig getid /Node:mynode/Server:server1/]
$AdminConfig getObjectType $server
```

- Using Jython:

```
server = AdminConfig.getid('/Node:mynode/Server:server1/')
print AdminConfig.getObjectType(server)
```

getSaveMode

Use the **getSaveMode** command to return the mode that is used when you invoke a **save** command. The command returns one of the following possible values:

- overwriteOnConflict - Saves changes even if they conflict with other configuration changes
- rollbackOnConflict - Fails a save operation if changes conflict with other configuration changes. This value is the default.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the current save mode setting, as the following example displays:

```
rollbackOnConflict
```

Examples

- Using Jacl:

  ```
  $AdminConfig getSaveMode
  ```

- Using Jython:

  ```
  print AdminConfig.getSaveMode()
  ```

getValidationLevel

Use the **getValidationLevel** command to return the validation used when files are extracted from the repository.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the validation level, as the following example displays:

```
WASX7189I: Validation level set to HIGH
```

Examples

- Using Jacl:

  ```
  $AdminConfig getValidationLevel
  ```

- Using Jython:

  ```
  print AdminConfig.getValidationLevel()
  ```

getValidationSeverityResult

Use the **getValidationSeverityResult** command to return the number of validation messages with the given severity from the most recent validation.

Target object

None.

Required parameters

severity
Specifies which severity level for which to return the number of validation messages. Specify an integer value between 0 and 9.

Optional parameters

None.

Sample output

The command returns a string that indicates the number of validation messages of the given severity, as the following example displays:

```
16
```

Examples

- Using Jacl:

  ```
  $AdminConfig getValidationSeverityResult 1
  ```

- Using Jython:

  ```
  print AdminConfig.getValidationSeverityResult(1)
  ```

hasChanges

Use the **hasChanges** command to determine if unsaved configuration changes exist.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns `1` if unsaved configuration changes exist or `0` if unsaved configuration changes do not exist, as the following example displays:

```
1
```

Examples

- Using Jacl:

  ```
  $AdminConfig hasChanges
  ```

- Using Jython:

  ```
  print AdminConfig.hasChanges()
  ```

help

Use the **help** command to display static help information for the AdminConfig object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of options for the help command, as the following example displays:

```
WASX7053I: The AdminConfig object communicates with the configuration service in a WebSphere Application Server
to manipulate configuration data for an Application Server installation.  The AdminConfig object has commands to list,
create, remove, display, and modify configuration data, as well as commands to display information about configuration data types.

Most of the commands supported by the AdminConfig object operate in two modes: the default mode is one in which the AdminConfig
object communicates with the Application Server to accomplish its tasks. A local mode is also possible, in which no server
communication takes place.  The local mode of operation is invoked by bringing up the scripting client without a server connected
using the command line "-conntype NONE" option or setting the "com.ibm.ws.scripting.connectionType=NONE" property in the
wsadmin.properties file.

The following commands are supported by the AdminConfig object; more detailed information about each of these commands is
available by using the help command of the AdminConfig object and by supplying the name of the command as an argument.

attributes               Shows the attributes for a given type
checkin                  Checks a file into the configuration repository.
convertToCluster         Converts a server to be the first member of a new server cluster
create                   Creates a configuration object, given a type, a parent, and a list of attributes, and
               optionally an attribute name for the new object
createClusterMember        Creates a new server that is a member of an existing cluster.
createDocument           Creates a new document in the configuration repository.
installResourceAdapter       Installs a J2C resource adapter with the given RAR file name and an option string in the node.
createUsingTemplate        Creates an object using a particular template type.
defaults                 Displays the default values for the attributes of a given type.
deleteDocument           Deletes a document from the configuration repository.
existsDocument           Tests for the existence of a document in the configuration repository.
extract                  Extracts a file from the configuration repository.
getCrossDocumentValidationEnabled  Returns true if cross-document validation is enabled.
getid                    Show the configuration ID of an object, given a string version of its containment
getObjectName            Given a configuration ID, returns a string version of the ObjectName
               for the corresponding running MBean, if any.
getSaveMode              Returns the mode used when "save" is invoked
getValidationLevel        Returns the validation that is used when files are extracted from the repository.
getValidationSeverityResult    Returns the number of messages of a given severity from the most recent validation.
hasChanges               Returns true if unsaved configuration changes exist
help                     Shows help information
list                     Lists all the configuration objects of a given type
listTemplates            Lists all the available configuration templates of a given type.
modify                   Changes the specified attributes of a given configuration object
parents                  Shows the objects which contain a given type
queryChanges             Returns a list of unsaved files
remove                   Removes the specified configuration object
required                 Displays the required attributes of a given type.
reset                    Discards the unsaved configuration changes
save                     Commits the unsaved changes to the configuration repository
setCrossDocumentValidationEnabled  Sets the cross-document validation enabled mode.
setSaveMode              Changes the mode used when "save" is invoked
setValidationLevel        Sets the validation used when files are extracted from the repository.
show                     Shows the attributes of a given configuration object
showall                  Recursively shows the attributes of a given configuration
                         object, and all the objects that are contained within each attribute.
showAttribute            Displays only the value for the single attribute that is specified.
types                    Shows the possible types for configuration
validate                 Invokes validation
```

Examples

- Using Jacl:

  ```
  $AdminConfig help
  ```

- Using Jython:

  ```
  print AdminConfig.help()
  ```

installResourceAdapter

Use the **installResourceAdapter** command to install a Java 2 Connector (J2C) resource adapter with the given Resource Adapter Archive (RAR) file name and an option string in the node. When you edit the installed application with the embedded RAR, only existing J2C connection factory, J2C activation specs, and J2C administrative objects will be edited. No new J2C objects will be created.

Target object

None.

Required parameters

node
Specifies the node of interest.

RAR file name
Specifies the fully qualified file name of the RAR file that resides in the node that you specify.

Optional parameters

options
Specifies additional options for installing a resource adapter. The valid options include the following options:

- `rar.name`
- `rar.desc`
- `rar.archivePath`
- `rar.classpath`
- `rar.nativePath`
- `rar.threadPoolAlias`
- `rar.propertiesSet`

The rar.name option is the name for the J2C resource adapter. If you do not specify this option, the display name in the RAR deployment descriptor is used. If that name is not specified, the RAR file name is used. The `rar.desc` option is a description of the J2CResourceAdapter.

The `rar.archivePath` is the name of the path where you extract the file. If you do not specify this option, the archive is extracted to the `$\{CONNECTOR_INSTALL_ROOT\}` directory. The rar.classpath option is the additional class path.

rar.propertiesSet is constructed with the following:

```
name String
value String
type String
*desc String
*required true/false
* means the item is optional
```

Each attribute of the property are specified in a set of {}. A property is specified in a set of {}. You can specify multiple properties in `{}`.

Sample output

The command returns the configuration ID of the new J2CResourceAdapter object:

```
myResourceAdapter(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

Examples

- Using Jacl:

- Using Jython:

list

Use the **list** command to return a list of objects of a given type, possibly scoped by a parent. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

object type
Specifies the name of the object type. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

pattern
Specifies additional search query information using wildcard characters of Java regular expressions.

Optional parameters

None.

Sample output

The command returns a list of objects:

Examples

The following examples list each JDBC provider configuration object:

- Using Jacl:
```
$AdminConfig list JDBCProvider
```
- Using Jython:
```
print AdminConfig.list('JDBCProvider')
```

The following examples list each JDBC provider configuration object that begin with the `derby` string:

- Using Jacl:
```
$AdminConfig list JDBCProvider derby*
```
- Using Jython:
```
print AdminConfig.list('JDBCProvider derby*')
```

You can use regular Java expression patterns and wildcard patterns to specify command name for $AdminConfig list, types and listTemplates functions.

The following examples list the configuration objects of type `server` starting from `server1`:

- Using Jacl and regular Java expression patterns:
```
$AdminConfig list Server server1.*
```
- Using Jacl and wildcard patterns:
```
$AdminConfig list Server server1*
```
- Using Jython and regular Java expression patterns::
```
print AdminConfig.list("Server", "server1.*")
```
- Using Jython and wildcard patterns:

```
print AdminConfig.list("Server", "server1*")
```

The following examples list each find configuration object of that starts with `SSLConfig`:

- Using Jacl and regular Java expression patterns:
  ```
  $AdminConfig types SSLConfig.*
  ```
- Using Jacl and wildcard patterns:
  ```
  $AdminConfig types SSLConfig*
  ```
- Using Jython and regular Java expression patterns:
  ```
  print AdminConfig.types("SSLConfig.*")
  ```
- Using Jython and wildcard patterns:
  ```
  print AdminConfig.types("SSLConfig*")
  ```

listTemplates

Use the **listTemplates** command to display a list of template object IDs. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

object type
Specifies the name of the object type. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

pattern
Specifies additional search query information using wildcard characters of Java regular expressions.

Optional parameters

None.

Sample output

The example displays a list of all the JDBCProvider templates that are available on the system:

Examples

The following examples return each JDBC provider template:

- Using Jacl:
  ```
  $AdminConfig listTemplates JDBCProvider
  ```
- Using Jython:
  ```
  print AdminConfig.listTemplates('JDBCProvider')
  ```

The following examples return each JDBC provider template that begins with the `sybase` string:

- Using Jacl:
  ```
  $AdminConfig listTemplates JDBCProvider sybase*
  ```
- Using Jython:
  ```
  print AdminConfig.listTemplates('JDBCProvider sybase*')
  ```

modify

Use the **modify** command to support the modification of object attributes.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object to modify.

attributes
Specifies the attributes to modify for the configuration ID of interest.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminConfig modify ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
GenericJMSConnectionFactory_1) {{userID newID} {password newPW}}
```

- Using Jython with string attributes:

```
AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
GenericJMSConnectionFactory_1)', '[[userID newID] [password newPW]]')
```

- Using Jython with object attributes:

```
AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
GenericJMSConnectionFactory_1)', [['userID', 'newID'], ['password', 'newPW']])
```

parents

Use the **parents** command to obtain information about object types.

Target object

None.

Required parameters

object type
Specifies the object type of interest. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The example displays a list of object types:

```
Cell
Node
Server
```

Examples

- Using Jacl:
  ```
  $AdminConfig parents JDBCProvider
  ```
- Using Jython:
  ```
  print AdminConfig.parents('JDBCProvider')
  ```

queryChanges

Use the **queryChanges** command to return a list of unsaved configuration files.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The example displays a string that contains a list of files with unsaved changes:

```
WASX7146I: The following configuration files contain unsaved changes:
cells/mycell/nodes/mynode/servers/server1|resources.xml
```

Examples

- Using Jacl:
  ```
  $AdminConfig queryChanges
  ```
- Using Jython:
  ```
  print AdminConfig.queryChanges()
  ```

remove

Use the **remove** command to remove a configuration object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration object of interest.

Optional parameters

None.

Examples

- Using Jacl:
  ```
  $AdminConfig remove ds1(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_6)
  ```
- Using Jython:

```
AdminConfig.remove('ds1(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_6)')
```

required

Use the **required** command to display the required attributes that are contained by an object of a certain type.

Target object

None.

Required parameters

type
Specifies the object type for which to display the required attributes. The name of the object type is based on the XML configuration files. It does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The example displays a string that contains a list of the required attributes with its type:

```
Attribute                      Type
streamHandlerClassName         String
protocol                       String
```

Examples

- Using Jacl:

  ```
  $AdminConfig required URLProvider
  ```

- Using Jython:

  ```
  print AdminConfig.required('URLProvider')
  ```

reset

Use the **reset** command to reset the temporary workspace that holds updates to the configuration.

Target object

None.

Required parameters

None.

Optional parameters

None.

Examples

- Using Jacl:

  ```
  $AdminConfig reset
  ```

- Using Jython:

```
AdminConfig.reset()
```

resetAttributes

Use the **resetAttributes** command to reset specific attributes for the configuration object of interest.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the configuration object of interest.

attributes
Specifies the attribute to reset and the value to which the attribute is reset.

Optional parameters

None.

Examples

- Using Jacl:

```
set ds [$AdminConfig getid /DataSource:myDS]
$AdminConfig resetAttributes $ds {{"classpath" "c:/temp/testing;c:/temp/test"}}
```

- Using Jython:

```
ds = AdminConfig.getid("/DataSource:myDS")
AdminConfig.resetAttributes(ds, [["classpath", "c:/temp/testing;c:/temp/test"]])
```

save

Use the **save** command to save changes to the configuration repository.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The **save** command does not return output.

Examples

- Using Jacl:

```
$AdminConfig save
```

- Using Jython:

```
AdminConfig.save()
```

setCrossDocumentValidationEnabled

Use the **setCrossDocumentValidationEnabled** command to set the cross-document validation enabled mode. Values include `true` or `false`.

Target object

None.

Required parameters

flag
Specifies whether cross-document validation is enabled or disabled. Specify `true` to enable or `false` to disable cross-document validation.

Optional parameters

None.

Sample output

The command returns a status statement for cross-document validation, as the following example displays:

```
WASX7188I: Cross-document validation enablement set to true
```

Examples

- Using Jacl:

  ```
  $AdminConfig setCrossDocumentValidationEnabled true
  ```

- Using Jython:

  ```
  print AdminConfig.setCrossDocumentValidationEnabled('true')
  ```

setSaveMode

Use the **setSaveMode** command to modify the behavior of the **save** command.

Target object

None.

Required parameters

save mode
Specifies the save mode to use. The default value is `rollbackOnConflict`. When the system discovers a conflict while saving, the unsaved changes are not committed. The alternative value is `overwriteOnConflict`, which saves the changes to the configuration repository even if conflicts exist. To use `overwriteOnConflict` as the value of this command, the deployment manager must be enabled for configuration overwrite.

Optional parameters

None.

Sample output

The **setSaveMode** command does not return output.

Examples

- Using Jacl:
  ```
  $AdminConfig setSaveMode overwriteOnConflict
  ```
- Using Jython:
  ```
  AdminConfig.setSaveMode('overwriteOnConflict')
  ```

setValidationLevel

Use the **setValidationLevel** command to set the validation that is used when files are extracted from the repository.

Target object

None.

Required parameters

level
Specifies the validation to use. Five validation levels are available: `none`, `low`, `medium`, `high`, or `highest`.

Optional parameters

None.

Sample output

The command returns a string that contains the validation level setting, as the following example displays:

```
WASX7189I: Validation level set to HIGH
```

Examples

- Using Jacl:
  ```
  $AdminConfig setValidationLevel high
  ```
- Using Jython:
  ```
  print AdminConfig.setValidationLevel('high')
  ```

show

Use the **show** command to return the top-level attributes of the given object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

Optional parameters

None.

Sample output

The command returns a string that contains the attribute value, as the following example displays:

```
[name "Sample Datasource"] [description "Data source for the Sample entity beans"]
```

Examples

- Using Jacl:

```
$AdminConfig show Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xmlJDBCProvider_1)
```

- Using Jython:

```
print AdminConfig.show('Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xmlJDBCProvider_1)')
```

showall

Use the **showall** command to recursively show the attributes of a given configuration object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

Optional parameters

None.

Sample output

The command returns a string that contains the attribute value, as the following example displays:

```
[datasourceHelperClassname com.ibm.websphere.rsadapter.DerbyDataStoreHelper]
[description "Datasource for the WebSphere Default Application"]
[jndiName DefaultDatasource]
[name "Default Datasource"]
[propertySet [[resourceProperties [[[description "Location of Apache Derby default database."]
[name databaseName]
[type string]
[value ${WAS_INSTALL_ROOT}/bin/DefaultDB]] [[name remoteDataSourceProtocol]
[type string]
[value []]] [[name shutdownDatabase]
[type string]
[value []]] [[name dataSourceName]
[type string]
[value []]] [[name description]
[type string]
[value []]] [[name connectionAttributes]
[type string]
[value []]] [[name createDatabase]
[type string]
[value []]]]]]]
[provider "Apache Derby JDBC Driver(cells/pongo/nodes/pongo/servers/server1|resources.xml#JDBCProvider_1)"]
[relationalResourceAdapter "WebSphere Relational Resource Adapter(cells/pongo/nodes/pongo/servers/server1|
resources.xml#builtin_rra)"]
[statementCacheSize 0]
```

Examples

- Using Jacl:

```
$AdminConfig showall "Default Datasource(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_1)
```

- Using Jython:

```
print AdminConfig.showall
  ("Default Datasource(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_1)")
```

showAttribute

Use the **showAttribute** command to display only the value for the single attribute that you specify.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

attribute
Specifies the attribute to query.

Optional parameters

None.

Sample output

The output of this command is different from the output of the **show** command when a single attribute is specified. The **showAttribute** command does not display a list that contains the attribute name and value. It only displays the attribute value, as the following example displays:

```
mynode
```

Examples

- Using Jacl:

```
set ns [$AdminConfig getid /Node:mynode/]
$AdminConfig showAttribute $ns hostName
```

- Using Jython:

```
ns = AdminConfig.getid('/Node:mynode/')
print AdminConfig.showAttribute(ns, 'hostName')
```

types

Use the **types** command to return a list of the configuration object types that you can manipulate. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of object types, as the following example displays:

```
AdminService
Agent
ApplicationConfig
ApplicationDeployment
ApplicationServer
AuthMechanism
AuthenticationTarget
```

```
AuthorizationConfig
AuthorizationProvider
AuthorizationTableImpl
BackupCluster
CMPConnectionFactory
CORBAObjectNameSpaceBinding
Cell
CellManager
Classloader
ClusterMember
ClusteredTarget
CommonSecureInteropComponent
```

Examples

The following examples return each object type in your configuration:

- Using Jacl:

  ```
  $AdminConfig types
  ```

- Using Jython:

  ```
  print AdminConfig.types()
  ```

The following examples return each object type in your configuration that contains the `security` string:

- Using Jacl:

  ```
  $AdminConfig types *security*
  ```

- Using Jython:

  ```
  print AdminConfig.types('*security*')
  ```

uninstallResourceAdapter

Use the **uninstallResourceAdapter** command to uninstall a Java 2 Connector (J2C) resource adapter with the given J2C resource adapter configuration ID and an option list. When you remove a J2CResourceAdapter object from the configuration repository, the installed directory will be removed at the time of synchronization. A stop request will be sent to the J2CResourceAdapter MBean that was removed.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the resource adapter to remove.

Optional parameters

options list
Specifies the uninstall options for command. The valid option is `force`. This option forces the uninstallation of the resource adapter without checking whether the resource adapter is being used by an application. The application that is using it will not be uninstalled. If you do not specify the force option and the specified resource adapter is still in use, the resource adapter is not uninstalled.

Sample output

The command returns the configuration ID of the J2C resource adapter that is removed, as the following example displays:

```
WASX7397I: The following J2CResourceAdapter objects are removed:
MyJ2CRA(cells/juniarti/nodes/juniarti|resources.xml#J2CResourceAdapter_1069433028609)
```

Examples

- Using Jacl:

```
set j2cra [$AdminConfig getid /J2CResourceAdapter:MyJ2CRA/]
$AdminConfig uninstallResourceAdapter $j2cra {-force}
$AdminConfig save
```

- Using Jython:

```
j2cra = AdminConfig.getid('/J2CResourceAdapter:MyJ2CRA/')
print AdminConfig.uninstallResourceAdapter(j2cra, '[-force]')
AdminConfig.save()
```

unsetAttributes

Use the **unsetAttributes** command to reset specific attributes for a configuration object to the default values.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the configuration object of interest.

attributes
Specifies the attributes to reset to the default values.

Optional parameters

None.

Examples

- Using Jacl:

```
set cluster [$AdminConfig getid /ServerCluster:myCluster]
$AdminConfig unsetAttributes $cluster {"enableHA", "preferLocal"}
```

- Using Jython:

```
cluster = AdminConfig.getid("/ServerCluster:myCluster")
AdminConfig.unsetAttributes(cluster, ["enableHA", "preferLocal"])
```

validate

Use the **validate** command to request the configuration validation results based on the files in your workspace, the value of the cross-document validation enabled flag, and the validation level setting. Optionally, you can specify a configuration ID to set the scope. If you specify a configuration ID, the scope of this request is the object named by the configuration ID parameter.

Target object

None.

Required parameters

None.

Optional parameters

configuration ID
Specifies the configuration ID of the object of interest.

Sample output

The command returns a string that contains the results of the validation, as the following example displays:

```
WASX7193I: Validation results are logged in c:\WebSphere5\AppServer\logs\wsadmin.valout: Total number of messages: 16
WASX7194I: Number of messages of severity 1: 16
```

Examples

- Using Jacl:

  ```
  $AdminConfig validate
  ```

- Using Jython:

  ```
  print AdminConfig.validate()
  ```

Commands for the AdminControl object

Use the AdminControl object to invoke operational commands that manage objects for the application server.

Many of the AdminControl commands have multiple signatures so that they can either invoke in a raw mode using parameters that are specified by Java Management Extensions (JMX), or by using strings for parameters. In addition to operational commands, the AdminControl object supports some utility commands for tracing, reconnecting with a server, and converting data types.

The following commands are available for the AdminControl object:

completeObjectName

Use the **completeObjectName** command to create a string representation of a complete ObjectName value that is based on a fragment. This command does not communicate with the server to find a matching ObjectName value. If the system finds several MBeans that match the fragment, the command returns the first one.

Target object

None.

Required parameters

object name
Specifies the name of the object to complete. (ObjectName)

template
Specifies the name of the template to use. For example, the template might be `type=Server,*`. (java.lang.String)

Optional parameters

None.

Sample output

The command does not return output.

Examples

- Using Jacl:
```
set serverON [$AdminControl completeObjectName node=mynode,type=Server,*]
```
- Using Jython:
```
serverON = AdminControl.completeObjectName('node=mynode,type=Server,*')
```

getAttribute

Use the **getAttribute** command to return the value of the attribute for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to query. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'DeploymentManager'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl getAttribute $objNameString processType
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.getAttribute(objNameString, 'processType')
```

getAttribute_jmx

Use the **getAttribute_jmx** command to return the value of the attribute for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to query. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'DeploymentManager'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
set objName [java::new javax.management.ObjectName $objNameString]
$AdminControl getAttribute_jmx $objName processType
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:=type=Server,*')
import  javax.management  as  mgmt
objName =  mgmt.ObjectName(objNameString)
print AdminControl.getAttribute_jmx(objName, 'processType')
```

getAttributes

Use the **getAttributes** command to return the attribute values for the names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the names of the attributes to query. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'[ [cellName myCell01] [nodeName myCellManager01] ]'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl getAttributes $objNameString "cellName nodeName"
```

- Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectname('WebSphere:type=Server,*)
print AdminControl.getAttributes(objNameString, '[cellName nodeName]')
```

- Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectname('WebSphere:type=Server,*)
print AdminControl.getAttributes(objNameString, ['cellName', 'nodeName'])
```

getAttributes_jmx

Use the **getAttributes_jmx** command to return the attribute values for the names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the names of the attributes to query. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns an attribute list.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attrs [java::new {String[]} 2 {cellName nodeName}]
$AdminControl getAttributes_jmx $objName $attrs
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
attrs = ['cellName', 'nodeName']
print AdminControl.getAttributes_jmx(objName, attrs)
```

getCell

Use the **getCell** command to return the name of the connected cell.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the cell name that you query, as the following example displays:

```
Mycell
```

Examples

- Using Jacl:

```
$AdminControl getCell
```

- Using Jython:

```
print AdminControl.getCell()
```

getConfigId

Use the **getConfigId** command to create a configuration ID from an ObjectName or an ObjectName fragment. Each MBean does not have corresponding configuration objects. If several MBeans correspond to an ObjectName fragment, a warning is created and a configuration ID builds for the first MBean that the system finds.

Target object

None.

Required parameters

object name
Specifies the name of the object of interest. The object name string can be a wildcard, specified with an asterisk character (*).

Optional parameters

None.

Sample output

The command returns a string that contains the configuration ID of interest.

Examples

- Using Jacl:
- Using Jython:

getDefaultDomain

Use the **getDefaultDomain** command to return the default domain name from the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the default domain name of interest, as the following example displays:

```
WebSphere
```

Examples

- Using Jacl:

  ```
  $AdminControl getDefaultDomain
  ```

- Using Jython:

```
print AdminControl.getDefaultDomain()
```

getDomainName

Use the **getDomainName** command to return the domain name from the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the domain name of interest, as the following example displays:

```
WebSphere
```

Examples

- Using Jacl:

  ```
  $AdminControl getDomainName
  ```

- Using Jython:

  ```
  print AdminControl.getDomainName()
  ```

getHost

Use the **getHost** command to return the name of your host.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the name of the host of interest, as the following example displays:

```
myHost
```

Examples

- Using Jacl:

```
$AdminControl getHost
```

- Using Jython:

```
print AdminControl.getHost()
```

getMBeanCount

Use the **getMBeanCount** command to return the number of MBeans that are registered in the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns an integer value that contains the number of MBeans that are registered in the server, as the following example displays:

```
151
```

Examples

- Using Jacl:

```
$AdminControl getMBeanCount
```

- Using Jython:

```
print AdminControl.getMBeanCount()
```

getMBeanInfo_jmx

Use the **getMBeanInfo_jmx** command to return the Java Management Extension MBeanInfo structure that corresponds to an ObjectName value. No string signature exists for this command, because the Help object displays most of the information available from the **getMBeanInfo_jmx** command.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

Optional parameters

None.

Sample output

The command returns a javax.management.MBeanInfo object, as the following example displays:

```
javax.management.modelmbean.ModelMBeanInfoSupport@10dd5f35
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objectNameString]
$AdminControl getMBeanInfo_jmx $objName
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.getMBeanInfo_jmx(objName)
```

getNode

Use the **getNode** command to return the name of the connected node.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string containing the name of the node, as the following example displays:

```
myNode01
```

Examples

- Using Jacl:

```
$AdminControl getNode
```

- Using Jython:

```
print AdminControl.getNode()
```

getObjectInstance

Use the **getObjectInstance** command to return the object instance that matches the input object name.

Target object

None.

Required parameters

object name
Specifies the name of the object of interest. (ObjectName)

Optional parameters

None.

Sample output

The command returns the object instance that matches the input object name, as the following example displays:

```
javax.management.modelmbean.RequiredModelMBean
```

Examples

- Using Jacl:

  ```
  set server [$AdminControl completeObjectName type=Server,*]
  set serverOI [$AdminControl getObjectInstance $server]
  ```

  Use the following example to manipulate the return value of the getObjectInstance command:

  ```
  puts [$serverOI getClassName]
  ```

- Using Jython:

  ```
  server = AdminControl.completeObjectName('type=Server,*')
  serverOI = AdminControl.getObjectInstance(server)
  ```

  Use the following example to manipulate the return value of the getObjectInstance command:

  ```
  print serverOI.getClassName()
  ```

getPort

Use the **getPort** command to return the name of the port used for the scripting connection.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the port number of the port that the system uses to establish the scripting connection, as the following example displays:

```
8877
```

Examples

- Using Jacl:

  ```
  $AdminControl getPort
  ```

- Using Jython:

  ```
  print AdminControl.getPort()
  ```

getPropertiesForDataSource (Deprecated)

The **getPropertiesForDataSource** command is deprecated, and no replacement exists. This command incorrectly assumes the availability of a configuration service when running in connected mode.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the following message:

```
WASX7389E: Operation not supported - getPropertiesForDataSource command is not supported.
```

Examples

- Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]
$AdminControl getPropertiesForDataSource $ds
```

- Using Jython:

```
ds = AdminConfig.list('DataSource')

# get line separator
import  java.lang.System  as  sys
lineSeparator = sys.getProperty('line.separator')

dsArray = ds.split(lineSeparator)
print AdminControl.getPropertiesForDataSource(dsArray[0])
```

getType

Use the **getType** command to return the connection type used for the scripting connection.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the connection type for the scripting connection, as the following example displays:

```
SOAP
```

Examples

- Using Jacl:

```
$AdminControl getType
```

- Using Jython:

```
print AdminControl.getType()
```

help

Use the **help** command to return general help text for the AdminControl object.

Target object

None.

Required parameters

None.

Optional parameters

command
Specifies the command for which to return help information. The command name is not case-sensitive.

Sample output

The command returns a string that details specific options for the **help** command, as the following example displays:

```
WASX7027I: The AdminControl object enables the manipulation of MBeans that run in a WebSphere
Application Server process. The number and type of MBeans that are available to the scripting client depend
on the server to which the client is connected.  If the client is connected to a deployment manager, then
all the MBeans running in the Deployment Manager are visible, as are all the MBeans running in the node agents
that are connected to this deployment manager, and all the MBeans that run in the application servers
on those nodes.

The following commands are supported by the AdminControl object; more detailed information about each of these
commands is available by using the "help" command of the AdminControl object and supplying the name
of the command as an argument.

Many of these commands support two different sets of signatures: one that accepts and returns strings, and one
low-level set that works with JMX objects like ObjectName and AttributeList. In most situations, the string
signatures are likely to be more useful, but JMX-object signature versions are supplied as well.  Each of these
JMX-object signature commands has "_jmx" appended to the command name, so an "invoke" command, as well as a
"invoke_jmx" command are supported.

completeObjectName  Return a String version of an object name given atemplate name
getAttribute_jmx   Given ObjectName and name of attribute, returns value ofattribute
getAttribute       Given String version of ObjectName and name of attribute, returns value of attribute
getAttributes_jmx   Given ObjectName and array of attribute names, returns AttributeList
getAttributes      Given String version of ObjectName and attribute names, returns String of name value pairs
getCell            returns the cell name of the connected server
getConfigId        Given String version of ObjectName, return a config id for the corresponding configuration
          object, if any.
getDefaultDomain   returns "WebSphere"
getDomainName      returns "WebSphere"
getHost            returns String representation of connected host
getMBeanCount      returns number of registered beans
getMBeanInfo_jmx   Given ObjectName, returns MBeanInfo structure for MBean
getNode            returns the node name of the connected server
getPort            returns String representation of port in use
getType            returns String representation of connection type in use help  help         Show help information
invoke_jmx         Given ObjectName, name of command, array of parameters andsignature, invoke command on MBean specifie
invoke             Invoke a command on the specified MBean
isRegistered_jmx   true if supplied ObjectName is registered
isRegistered       true if supplied String version of ObjectName is registered
makeObjectName     Return an ObjectName built with the given string
queryNames_jmx     Given ObjectName and QueryExp, retrieves set of ObjectNames that match.
queryNames         Given String version of ObjectName, retrieves String of ObjectNames that match.
reconnect          reconnects with serversetAttribute_jmx Given ObjectName and Attribute object, set attribute for
          MBean specified
```

```
setAttribute       Given String version of ObjectName, attribute name andattribute value, set attribute for MBean specified
setAttributes_jmx   Given ObjectName and AttributeList object, set attributes for the MBean specified
startServer        Given the name of a server, start that server.
stopServer         Given the name of a server, stop that server.
testConnection     Test the connection to a DataSource object
trace              Set the wsadmin trace specification
```

If you specify a specific command with the help command, the wsadmin tool returns detailed help about the command, as the following example displays:

```
WASX7043I: command: getAttribute
Arguments: object name, attribute
Description: Returns value of "attribute" for the MBean described by "object name."
```

Examples

- Using Jacl:

  ```
  $AdminControl help
  $AdminControl help getAttribute
  ```

- Using Jython:

  ```
  print AdminControl.help()
  print AdminControl.help('getAttribute')
  ```

invoke

Use the **invoke** command to invoke an object operation with or without parameters. The command invokes the object operation using the parameter list that you supply. The signature generates automatically. The types of parameters are supplied by examining the MBeanInfo that the MBean supplies. Returns the string result of the invocation. The string that is returned is controlled by the Mbean method that you invoked. If the MBean method is synchronous, then control is returned back to the wsadmin tool only when the operation is complete. If the Mbean method is asynchronous, control is returned back to the wsadmin tool immediately even though the invoked task might not be complete.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest.

operation
Specifies the operation to invoke.

Optional parameters

arguments
Specifies the arguments required for the operation. If no arguments are required for the operation of interest, you can omit the arguments parameter.

Sample output

The command returns a string that shows the result of the invocation.

Examples

- Using Jacl:

  ```
  set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
  $AdminControl invoke $objNameString stop
  ```

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled

set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled java.lang.String

set objNameString [$AdminControl completeObjectName WebSphere:type=DynaCache,*]
$AdminControl invoke $mbean getCacheStatistics {"DiskCacheSizeInMB ObjectsReadFromDisk4000K RemoteObjectMisses"}
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.invoke(objNameString, 'stop')

objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.invoke(objNameString, 'appendTraceString', 'com.ibm.*=all=enabled')

objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.invoke(objNameString, 'appendTraceString', 'com.ibm.*=all=enabled',  'java.lang.String')

objNameString = AdminControl.completeObjectName("WebSpheretype=DynaCache,*")
 AdminControl.invoke(dc, "getCacheStatistics", '["DiskCacheSizeInMB ObjectReadFromDisk4000K RemoteObjectMisses"]')
```

- Using Jython list:

```
objNameString = AdminControl.completeObjectName("WebSphere:type=DynaCache,*")
AdminControl.invoke(dc, "getCacheStatistics", [["DiskCacheSizeInMB", "ObjectReadFromDisk4000K", "RemoteObjectMisses"]]
```

invoke_jmx

Use the **invoke_jmx** command to invoke the object operation by conforming the parameter list to the signature. The command returns the result of the invocation.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

operation
Specifies the operation to invoke. (java.lang.String)

Optional parameters

arguments
Specifies the arguments required for the operation. If no arguments are required for the operation of interest, you can omit the arguments parameter. (java.lang.String[] or java.lang.Object[])

Sample output

The command returns a string that shows the result of the invocation.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [java::new javax.management.ObjectName $objNameString]
set parms [java::new {java.lang.Object[]} 1 com.ibm.ejs.sm.*=all=disabled]
set signature [java::new {java.lang.String[]} 1 java.lang.String]
$AdminControl invoke_jmx $objName appendTraceString $parms $signature
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName =  mgmt.ObjectName(objNameString)
```

```
parms = ['com.ibm.ejs.sm.*=all=disabled']
signature = ['java.lang.String']
print AdminControl.invoke_jmx(objName, 'appendTraceString', parms, signature)
```

isRegistered

Use the **isRegistered** command to determine if a specific object name is registered.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a boolean value for the object of interest. If the ObjectName value is registered in the server, then the value is `true`, as the following example displays:

```
true
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl isRegistered $objNameString
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.isRegistered(objNameString)
```

isRegistered_jmx

Use the **isRegistered_jmx** command to determine if a specific object name is registered.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a boolean value for the object of interest. If the ObjectName value is registered in the server, then the value is `true`, as the following example displays:

```
true
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objNameString]
$AdminControl isRegistered_jmx $objName
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.isRegistered_jmx(objName)
```

makeObjectName

Use the **makeObjectName** command to create an ObjectName value that is based on the strings input. This command does not communicate with the server, so the ObjectName value that results might not exist. If the string you supply contains an extra set of double quotes, they are removed. If the string does not begin with a Java Management Extensions (JMX) domain, or a string followed by a colon, then the WebSphere Application Server string appends to the name.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns an Objectname object constructed from the object name string.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,node=mynode,*]
set objName [$AdminControl makeObjectName $objNameString]
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,node=mynode,*')
objName = AdminControl.makeObjectName(objectNameString)
```

queryMBeans

Use the **queryMBeans** command to query for a list of object instances that match the object name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (ObjectName)

Optional parameters

query
Specifies the query expression. (QueryExp)

Sample output

The command returns a list of object instances for the object name specified, as the following example displays:

```
WebSphere:name=PlantsByWebSphere,process=server1,platform=dynamicproxy,node=Goodog,
J2EEName=PlantsByWebSphere,Server=server1,version=6.1.0.0,type=Application,
mbeanIdentifier=cells/GooddogNode02Cell/applications/PlantsByWebSphere.ear/
deployments/PlantsByWebSphere/deployment.xml#ApplicationDeployment_1126623343902,
cell=GooddogNode02Cell
```

Examples

- Using Jacl:

  ```
  set apps [$AdminControl queryMBeans type=Application,*]
  ```

  Use the following example to manipulate the return value of the queryMBeans command:

  ```
  set appArray [$apps toArray]
  set app1 [java::cast javax.management.ObjectInstance [$appArray get 0]]
  puts [[$app1 getObjectName] toString]
  ```

  The following example specifies the object name and the query expression:

  ```
  set apps [$AdminControl queryMBeans =type=Application,* [java::null]]
  ```

  Use the following example to manipulate the return value of the queryMBeans command:

  ```
  set appArray [$apps toArray]
  set app1 [java::cast javax.management.ObjectInstance [$appArray get 0]]
  puts [[$app1 getObjectName] toString]
  ```

- Using Jython:

  ```
  apps = AdminControl.queryMBeans('type=Application,*')
  ```

  Use the following example to manipulate the return value of the queryMBeans command:

  ```
  appArray = apps.toArray()
  app1 = appArray[0]
  print app1.getObjectName().toString()
  ```

  The following example specifies the object name and the query expression:

  ```
  apps = AdminControl.queryMBeans('type=Application,*',None)
  ```

  Use the following example to manipulate the return value of the queryMBeans command:

  ```
  appArray = apps.toArray()
  app1 = appArray[0]
  print app1.getObjectName().toString()
  ```

queryNames

Use the **queryNames** command to query for a list of each of the ObjectName objects based on the name template.

Target object

None.

Required parameters

object name
Specifies the object name of interest. You can specify a wildcard for the object name parameter with the asterisk character (*). (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the ObjectNames that match the input object name, as the following example displays:

```
WebSphere:cell=BaseApplicationServerCell,
name=server1,mbeanIdentifier=server1,
type=Server,node=mynode,process=server1
```

Examples

- Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Server,*
  ```

- Using Jython:

  ```
  print AdminControl.queryNames('WebSphere:type=Server,*')
  ```

queryNames_jmx

Use the **queryNames_jmx** command to query for a list of each of the ObjectName objects based on the name template and the query conditions that you specify.

Target object

None.

Required parameters

object name
Specifies the object name of interest. You can specify a wildcard for the object name parameter with the asterisk character (*). (ObjectName)

query
Specifies the query expression to use. (javax.management.QueryExp)

Optional parameters

None.

Sample output

The command returns a string that contains the ObjectNames that match the input object name, as the following example displays:

```
[WebSphere:cell=BaseApplicationServerCell,name=server1,mbeanIdentifier=
server1,type=Server,node=mynode,process=server1]
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objNameString]
set null [java::null]
$AdminControl queryNames_jmx $objName $null
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.queryNames_jmx(objName, None)
```

reconnect

Use the **reconnect** command to reconnect to the server, and to clear information out of the local cache.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a message that displays the status of the operation, as the following example displays:

```
WASX7074I: Reconnect of SOAP connector to host myhost completed.
```

Examples

- Using Jacl:

```
$AdminControl reconnect
```

- Using Jython:

```
print AdminControl.reconnect()
```

setAttribute

Use the **setAttribute** command to set the attribute value for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (java.lang.String)

attribute name
Specifies the name of the attribute to set. (java.lang.String)

attribute value
Specifies the value of the attribute of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns does not return output.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
$AdminControl setAttribute $objNameString traceSpecification com.ibm.*=all=disabled
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
print AdminControl.setAttribute(objNameString, 'traceSpecification',  'com.ibm.*=all=disabled')
```

setAttribute_jmx

Use the **setAttribute_jmx** command to set the attribute value for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to set. (Attribute)

Optional parameters

None.

Sample output

The command returns does not return output.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute traceSpecification com.ibm.*=all=disabled]
$AdminControl setAttribute_jmx $objName $attr
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName = AdminControl.makeObjectName(objectNameString)
attr = mgmt.Attribute('traceSpecification', 'com.ibm.*=all=disabled')
print AdminControl.setAttribute_jmx(objName, attr)
```

setAttributes

Use the **setAttributes** command to set the attribute values for the object names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (String)

attributes
Specifies the names of the attributes to set. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns a list of object names that are successfully set by the command invocation, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TracesService,*]
$AdminControl setAttributes $objNameString {{traceSpecification com.ibm.ws.*=all=enabled}}
```

- Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TracesService,*')
AdminControl.setAttributes(objNameString, '[[traceSpecification "com.ibm.ws.*=all=enabled"]]')
```

- Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TracesService,*')
print AdminControl.setAttributes(objNameString, [['traceSpecification', 'com.ibm.ws.*=all=enabled']])
```

setAttributes_jmx

Use the **setAttributes_jmx** command to set the attribute values for the object names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (String)

attributes
Specifies the names of the attributes to set. (javax.management.AttributeList)

Optional parameters

None.

Sample output

The command returns an attribute list of object names that are successfully set by the command invocation, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute traceSpecification com.ibm.ws.*=all=enabled]
set alist [java::new javax.management.AttributeList]
$alist add $attr
$AdminControl setAttributes_jmx $objName $alist
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName = AdminControl.makeObjectName(objectNameString)
attr = mgmt.Attribute('traceSpecification', 'com.ibm.ws.*=all=enabled')
alist = mgmt.AttributeList()
alist.add(attr)
print AdminControl.setAttributes_jmx(objName, alist)
```

startServer

Use the **startServer** command to start the specified application server by locating it in the configuration. This command uses the default wait time. Use the following guidelines to determine which parameters to use:

- If the scripting process is attached to a node agent server, you must specify the server name. You can also specify the optional wait time and node name parameters.
- If the scripting process is attached to a deployment manager process, you must specify the server name and node name. You can also specify the optional wait time parameter.

Target object

None.

Required parameters

server name
Specifies the name of the server to start. (java.lang.String)

Optional parameters

node name
Specifies the name of the node of interest. (java.lang.String)

wait time
Specifies the number of seconds that the start process waits for the server to start. The default wait time is 1200 seconds. (java.lang.String)

Sample output

The command returns a message to indicate if the server starts successfully, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

Using Jacl:

- The following example specifies only the name of the server to start:

```
$AdminControl startServer server1
```

- The following example specifies the name of the server to start and the wait time:

```
$AdminControl startServer server1 100
```

- The following example specifies the name of the server to start and the name of the node:

```
$AdminControl startServer server1 myNode
```

- The following example specifies the name of the server, the name of the node, and the wait time:

```
$AdminControl startServer server1 myNode 100
```

Using Jython:

- The following example specifies only the name of the server to start:

```
AdminControl.startServer('server1')
```

- The following example specifies the name of the server to start and the wait time:

```
AdminControl.startServer('server1', 100)
```

- The following example specifies the name of the server to start and the name of the node:

```
AdminControl.startServer('server1', 'myNode')
```

- The following example specifies the name of the server, the name of the node, and the wait time:

```
AdminControl.startServer('server1', 'myNode', 100)
```

stopServer

Use the **stopServer** command to stop the specified application server. When the **stopServer** command runs without the immediate or terminate flags, the server finishes any work in progress, but does not accept any new work once it begins the stop process. Use the following options to determine which parameters to use:

- Use the server name and the node name parameters to stop a server in a specific node.
- Use the server name and immediate flag parameters to stop the server immediately. If this parameter is not specified, the system stops the server normally.
- Use the server name, node name, and immediate flag parameters to immediately stop a server for a specific node.

Target object

None.

Required parameters

server name
Specifies the name of the server to start. (java.lang.String)

Optional parameters

node name
Specifies the name of the node of interest. (java.lang.String)

immediate flag
Specifies to stop the server immediately if the value is set to `immediate`. If you specify the immediate flag, the server does not finish processing any work in progress, does not accept any new work, and ends the server process. (java.lang.String)

terminate flag
Specifies that the server process should be terminated by the operating system. (String)

Sample output

The command returns a message to indicate if the server stops successfully, as the following example displays:

```
WASX7337I: Invoked stop for server "server1" Waiting for stop completion.
'WASX7264I: Stop completed for server "server1" on node "myNode"'
```

Examples

Using Jacl:

- The following example specifies only the name of the server to stop:
  ```
  $AdminControl stopServer server1
  ```
- The following example specifies the name of the server to stop and indicates that the server should stop immediately:
  ```
  $AdminControl stopServer server1 immediate
  ```
- The following example specifies the name of the server to stop and the name of the node:
  ```
  $AdminControl stopServer server1 myNode
  ```
- The following example specifies the name of the server, the name of the node, and indicates that the server should stop immediately:
  ```
  $AdminControl stopServer server1 myNode immediate
  ```

Using Jython:

- The following example specifies only the name of the server to stop:
  ```
  $AdminControl.stopServer('server1')
  ```
- The following example specifies the name of the server to stop and indicates that the server should stop immediately:
  ```
  $AdminControl.stopServer('server1','immediate')
  ```
- The following example specifies the name of the server to stop and the name of the node:
  ```
  $AdminControl.stopServer('server1','myNode')
  ```
- The following example specifies the name of the server, the name of the node, and indicates that the server should stop immediately:
  ```
  $AdminControl.stopServer('server1','myNode','immediate')
  ```

testConnection

Use the **testConnection** command to test a data source connection. This command works with the data source that resides in the configuration repository. If the data source to be tested is in the temporary workspace that holds the update to the repository, you must save the update to the configuration repository before running this command. Use this command with the configuration ID that corresponds to the data source and the WAS40DataSource object types.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the data source object of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a message that indicates a successful connection or a connection with a warning. If the connection fails, an exception is created from the server indicating the error. For example:

```
WASX7217I: Connection to provided datasource was successful.
```

Examples

- Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]
$AdminControl testConnection $ds
```

- Using Jython:

```
# get line separator
import java.lang.System as sys
lineSeparator = sys.getProperty('line.separator')
ds = AdminConfig.list('DataSource').split(lineSeparator)[0]
print AdminControl.testConnection(ds)
```

trace

Use the **trace** command to set the trace specification for the scripting process to the value that you specify.

Target object

None.

Required parameters

trace specification
Specifies the trace to enable for the scripting process. (java.lang.String)

Optional parameters

None.

Sample output

The command does not return output.

Examples

- Using Jacl:

```
$AdminControl trace com.ibm.ws.scripting.*=all=enabled
```

- Using Jython:

```
print AdminControl.trace('com.ibm.ws.scripting.*=all=enabled')
```

Commands for the AdminApp object

Use the AdminApp object to install, modify, and administer applications.

The AdminApp object interacts with the WebSphere Application Server management and configuration services to make application inquiries and changes. This interaction includes installing and uninstalling applications, listing modules, exporting, and so on.

You can start the scripting client when no server is running, if you want to use only local operations. To run in local mode, use the `-conntype NONE` option to start the scripting client. You receive a message that you are running in the local mode. Running the AdminApp object in local mode when a server is currently

running is not recommended. This is because any configuration changes made in local mode will not be reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager.

When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

The following commands are available for the AdminApp object:

deleteUserAndGroupEntries

Use the **deleteUserAndGroupEntries** command to delete users or groups for all roles, and to delete user IDs and passwords for all of the RunAs roles that are defined in the application.

Target object

None.

Required parameters

application name
Specifies the application of interest.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp deleteUserAndGroupEntries myapp
```

- Using Jython string:

```
print AdminApp.deleteUserAndGroupEntries('myapp')
```

- Using Jython list:

```
print AdminTask.deleteUserAndGroupEntries(['myapp'])
```

edit

Use the **edit** command to edit an application or module in batch mode. The **edit** command changes the application specified by the application name argument using the options specified by the options argument. No options are required for the **edit** command.

Target object

None.

Required parameters

application name
Specifies the application of interest.

options
Specifies the options to apply to the application or module configuration.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp edit "JavaMail Sample" {-MapWebModToVH {{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}}}
```

- Using Jython string:

```
print AdminApp.edit("JavaMail Sample", '[-MapWebModToVH [["JavaMail 32 Sample WebApp"
mtcomps.war,WEB-INF/web.xml newVH]]]')
```

- Using Jython list:

```
option = [["JavaMail 32 Sample WebApp", "mtcomps.war,WEB-INF/web.xml", "newVH"]]
mapVHOption = ["-MapWebModToVH", option]
print AdminApp.edit("JavaMail Sample", mapVHOption)
```

editInteractive

Use the **editInteractive** command to edit an application or module in interactive mode. The **editInteractive** command changes the application deployment. Specify these changes in the options parameter. No options are required for the **editInteractive** command.

Target object

None.

Required parameters

application name
Specifies the application of interest.

options
Specifies the options to apply to the application or module configuration.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp editInteractive ivtApp
```

- Using Jython string:

```
AdminApp.editInteractive('ivtApp')
```

export

Use the **export** command to export the application name parameter to a file that you specify by the file name.

Target object

None.

Required parameters

application name
Specifies the application of interest.

file name
Specifies the file name to export the application name to.

Optional parameters

exportToLocal
Specifies that the system should export the application of interest to the file name specified on the local client machine.

Examples

- Using Jacl:

```
$AdminApp export DefaultApplication c:/temp/export.ear {-exportToLocal}
```

- Using Jython:

```
AdminApp.export('DefaultApplication', 'c:/temp/export.ear', '[-exportToLocal]')
```

exportDDL

Use the **exportDDL** command to extract the data definition language (DDL) from the application name parameter to the directory name parameter that a directory specifies. The options parameter is optional.

Target object

None.

Required parameters

application name
Specifies the application of interest.

directory name
Specifies the name of the directory to export the application name to.

Optional parameters

options
Specifies the options to pass to the application name specified.

Examples

- Using Jacl:

```
$AdminApp exportDDL "My App" /usr/me/DDL {-ddlprefix myApp}
```

- Using Jython string:

```
print AdminApp.exportDDL("My App", '/usr/me/DDL', '[-ddlprefix myApp]')
```

exportFile

Use the **exportFile** command to export the contents of a single file specified by the uniform resource identifier (URI) from the application of interest.

Target object

None.

Required parameters

application name
Specifies the application of interest.

URI
Specifies the single file to export. Specify the URI within the context of an application, as the following example displays: `META-INF/application.xml`. To specify files within a module, the URI begins with a module URI, as the following example displays: `foo.war/WEB-INF/web.xml`.

filename
Specifies the fully qualified path and file name of the file to export to.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp exportFile "My App" myapp/components.jar/META-INF/ibm-ejb-jar-bnd.xml META-INF/ibm-ejb-jar-bnd.xml
```

- Using Jython string:

```
AdminApp.exportFile('My App', 'myapp/components.jar/META-INF/ibm-ejb-jar-bnd.xml', 'META-INF/ibm-ejb-jar-bnd.xml')
```

getDeployStatus

Use the **getDeployStatus** command to display the deployment status of the application. After installing or updating a large application, use this command to display detailed status information for application binary file expansion. You cannot start the application until the system extracts the application binaries.

Target object

None.

Required parameters

application name
Specifies the name of the application of interest.

Optional parameters

None.

Examples

- Using Jacl:
  ```
  $AdminApp getDeployStatus myApplication
  ```
- Using Jython:
  ```
  print AdminApp.getDeployStatus('myApplication')
  ```

help

Use the **help** command to display general help information about the AdminApp object.

Target object

None.

Required parameters

None.

Optional parameters

operation name
Specify this option to display help for an AdminApp command or installation option.

Sample output

The following output is returned if you do not specify an argument:

```
WASX7095I: The AdminApp object allows application objects to be manipulated including installing, uninstalling,
editing, and listing.  Most of the commands supported by AdminApp operate in two modes: the default mode is one
in which AdminApp communicates with the WebSphere Application Server to accomplish its tasks.  A local mode is also
possible, in which no server communication takes place.  The local mode of operation is invoked by including the
"-conntype NONE" flag in the option string supplied to the command.

The following commands are supported by AdminApp; more detailed information about each of these commands
is available by using the "help" command of AdminApp and supplying the name of the command as an argument.

edit               Edit the properties of an application
editInteractive    Edit the properties of an application interactively
export             Export application to a file
exportDDL          Extract DDL from application to a directory
help                Show help information
install            Installs an application, given a file name and an option string.
installInteractive  Installs an application in interactive mode, given a file name and an option string.
list               List all installed applications
listModules        List the modules in a specified
application options  Shows the options available, either for a given file, or in general.
taskInfo           Shows detailed information pertaining to a given installation task for a given file
uninstall          Uninstalls an application, given an application name and an option string
```

The following output is returned if you specify `uninstall` as the operation name argument:

```
WASX7102I: Method: uninstall
Arguments: application name, options
Description: Uninstalls application named by "application name" using the options supplied by String 2.
```

```
Method: uninstall
Arguments: application name
Description: Uninstalls the application specified by "application name" using default options.
```

Examples

Using Jacl:

- The following example does not specify any arguments:
  ```
  $AdminApp help
  ```
- The following example specifies the operation name argument:
  ```
  $AdminApp help uninstall
  ```

Using Jython:

- The following example does not specify any arguments:
  ```
  print AdminApp.help()
  ```
- The following example specifies the operation name argument:
  ```
  print AdminApp.help('uninstall')
  ```

install

Use the **install** command to install an application in non-interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.

Target object

None.

Required parameters

ear file
Specify the path of the .ear file to install.

Optional parameters

options
Specify the installation options for the command.

Examples

- Using Jacl:
  ```
  $AdminApp install c:/apps/myapp.ear
  ```
- Using Jython:
  ```
  print AdminApp.install('c:/apps/myapp.ear')
  ```

Many options are available for this command. You can obtain a list of valid options for an Enterprise Archive (EAR) file with the following command:

Using Jacl:

```
$AdminApp options myApp.ear
```

Using Jython:

```
print AdminApp.options('myApp.ear')
```

You can also obtain help for each object with the following command:

Using Jacl:

```
$AdminApp help MapModulesToServers
```

Using Jython:

```
print AdminApp.help('MapModulesToServers')
```

installInteractive

Use the **installInteractive** command to install an application in interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.

Target object

None.

Required parameters

ear file
Specify the path of the .ear file to install.

Optional parameters

options
Specify the installation options for the command.

Examples

- Using Jacl:

  ```
  $AdminApp installInteractive c:/websphere/appserver/installableApps/jmsample.ear
  ```

- Using Jython:

  ```
  print AdminApp.installInteractive('c:/websphere/appserver/installableApps/jmsample.ear')
  ```

isAppReady

Use the **isAppReady** command to determine if the specified application has been distributed and is ready to be run. Returns a value of `true` if the application is ready, or a value of `false` if the application is not ready. This command is not supported when the wsadmin tool is not connected to a server.

Target object

None.

Required parameters

application name
Specify the name of the application of interest.

Optional parameters

ignoreUnknownState
Tests to see if the specified application has been distributed and is ready to be run. Valid values for the ignoreUnknownState parameter include `true` and `false`. If you specify a value of `true`, nodes and servers with an unknown state will not be included in the final ready return. The command returns a value of `true` if the application is ready or a value of `false` if the application is not ready. This command is not supported when the wsadmin tool is not connected to a server.

Sample output

The following sample output is returned if you specify the application name parameter:

```
ADMA5071I: Distribution status check started for application DefaultApplication.WebSphere:cell=Node03Cell,
node=myNode,distribution=true
ADMA5011I: The cleanup of the temp directory for application DefaultApplication is complete.
ADMA5072I: Distribution status check completed for application DefaultApplication.true
```

The following sample output is returned if you specify the application name and ignoreUnknownState parameters:

```
ADMA5071I: Distribution status check started for application TEST.WebSphere:cell=myCell,node=myNode,distribution=unknown
ADMA5011I: The cleanup of the temp directory for application TEST is complete.
ADMA5072I: Distribution status check completed for application TEST.false
```

Examples

The following examples only specify the application name parameter:

- Using Jacl:

  ```
  $AdminApp isAppReady DefaultApplication
  ```

- Using Jython:

  ```
  print AdminApp.isAppReady('DefaultApplication')
  ```

The following examples specify the application name and ignoreUnknownState parameters:

- Using Jacl:

  ```
  $AdminApp isAppReady TEST true
  ```

- Using Jython:

  ```
  print AdminApp.isAppReady('TEST', 'true')
  ```

list

Use the **list** command to list the applications that are installed in the configuration.

Target object

None.

Required parameters

None.

Optional parameters

target
Lists the applications that are installed on a given target scope in the configuration.

Sample output

```
adminconsole
DefaultApplication
ivtApp
```

Examples

- Using Jacl:

  ```
  $AdminApp list
  ```

- Using Jython:

  ```
  print AdminApp.list()
  ```

The following examples specify a value for the target parameter:

- Using Jacl:

```
$AdminApp list WebSphere:cell=myCell,node=myNode,server=myServer
```

- Using Jython:

```
print AdminApp.list("WebSphere:cell=myCell,node=myNode,server=myServer")
```

listModules

Use the **listModules** command to list the modules in an application.

Target object

None.

Required parameters

application name
Specifies the application of interest.

Optional parameters

options
Specifies the list of application servers on which the modules are installed. The options parameter is optional. The valid option is -server.

Sample output

The following example is the concatenation of appname, #, module URI, +, and DD URI. You can pass this string to the **edit** and **editInteractive** AdminApp commands.

```
ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml
ivtApp#ivt_app.war+WEB-INF/web.xml
```

Examples

- Using Jacl:

```
$AdminApp listModules ivtApp
```

- Using Jython:

```
print AdminApp.listModules('ivtApp')
```

options

Use the **options** command to display a list of options for installing an Enterprise Archive (EAR) file.

Target object

None.

Required parameters

None.

Optional parameters

EAR file
Specifies the EAR file of interest.

application name
Specifies the application for which to display a list of options for editing an existing application.

application module name
Specifies the module name for which to display a list of options for editing a module in an existing application. This parameter requires the same module name format as the output that is returned by the **listModules** command.

file, operations
Displays a list of options for installing or updating an application or application module file. Specify one of the following valid values:

- installapp - Use this option to install the file that is specified.
- updateapp - Use this option to update an existing application with the file that is specified.
- addmodule - Use this option to add the module file that is specified to an existing application.
- updatemodule - Use this option to update an existing module in an application with the module file that is specified.

Sample output

```
WASX7112I: The following options are valid for "ivtApp"
MapRolesToUsers
BindJndiForEJBNonMessageBinding
MapEJBRefToEJB
MapWebModToVH
MapModulesToServers
distributeApp
nodistributeApp
useMetaDataFromBinary
nouseMetaDataFromBinary
createMBeansForResources
nocreateMBeansForResources
reloadEnabled
noreloadEnabled
verbose
installed.ear.destination
reloadInterval
```

Examples

The following example options command returns the valid options for an EAR file:

- Using Jacl:

```
$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear
```

- Using Jython:

```
print AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear')
```

The following example options command returns the valid options for an application:

- Using Jacl:

```
$AdminApp options ivtApp
```

- Using Jython:

```
print AdminApp.options('ivtApp')
```

The following example options command returns the valid options for an application module:

- Using Jacl:

```
$AdminApp options ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml
```

- Using Jython:

```
print AdminApp.options('ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml')
```

The following example options command returns the valid options for the operation that is requested with the input file:

- Using Jacl:

```
$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear updateapp
```

- Using Jython:

```
print AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear', 'updateapp')
```

publishWSDL

Use the **publishWSDL** command to publish Web Services Description Language (WSDL) files for the application that is specified in the application name parameter to the file that is specified in the file name parameter.

Target object

None.

Required parameters

file name
Specifies the file of interest.

application name
Specifies the application of interest

Optional parameters

SOAP address prefixes
Specifies the SOAP address prefixes to use.

Sample output

The publishWSDL command does not return output.

Examples

The following example publishWSDL command specifies the application name and the file name:

- Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlerServer c:/temp/a.zip
```

- Using Jython:

```
print AdminApp.publishWSDL('JAXRPCHandlerServer', 'c:/temp/a.zip')
```

The following example publishWSDL command specifies the application name, file name, and SOAP address prefixes parameter values:

- Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlersServer c:/temp/a.zip {{JAXRPCHandlersServerApp.war {{http http://localhost:9080}}}
```

- Using Jython:

```
print AdminApp.publishWSDL('JAXRPCHandlersServer', 'c:/temp/a.zip', '[[JAXRPCHandlersServerApp.war [[http http://local
```

searchJNDIReferences

Use the **searchJNDIReferences** command to list applications that refer to the Java Naming and Directory Interface (JNDI) name on a specific node.

Target object

None.

Required parameters

node configuration ID
Specifies the configuration ID for the node of interest.

Optional parameters

options
Specifies the options to use.

Sample output

```
WASX7410W: This operation may take a while depending on the number of applications installed in your system.
MyApp
MapResRefToEJB :ejb-jar-ic.jar : [eis/J2CCF1]
```

Examples

The following example assumes that an installed application named `MyApp` has a JNDI name of `eis/J2CCF1`:

- Using Jacl:

  ```
  $AdminApp searchJNDIReferences $node {-JNDIName eis/J2CCF1 -verbose}
  ```

- Using Jython:

  ```
  print AdminApp.searchJNDIReferences(node, '[-JNDIName eis/J2CCF1 -verbose]')
  ```

taskinfo

Use the **taskinfo** command to provide information about a particular task option for an application file. Many task names have changed between V5.x and V6.x for a similar or the exact same operation. You might need to update existing scripts if you are migrating from V5.x to V6.x.

Target object

None.

Required parameters

EAR file
Specifies the EAR file of interest.

task name
Specifies the task for which to request the information.

Optional parameters

None.

Sample output

```
MapWebModToVH: Selecting virtual hosts for Web modules
Specify the virtual host where you want to install the Web modules that are contained in
your application. Web modules can be installed on the same virtual host or dispersed among several hosts.
Each element of the MapWebModToVH task consists of the following three fields: "webModule," "uri," "virtualHost."
Of these fields, the following fields might be assigned new values: "virtualHost"and the following are
required: "virtualHost"

The current contents of the task after running default bindings are:
webModule: JavaMail Sample WebApp
uri: mtcomps.war,WEB-INF/web.xml
virtualHost: default_host
```

Examples

- Using Jacl:

```
$AdminApp taskInfo c:/websphere/appserver/installableApps/jmsample.ear MapWebModToVH
```

- Using Jython:

```
print AdminApp.taskInfo('c:/websphere/appserver/installableApps/jmsample.ear', 'MapWebModToVH')
```

uninstall

Use the **uninstall** command uninstall an existing application.

Target object

None.

Required parameters

application name
Specifies the name of the application to uninstall.

Optional parameters

None.

Sample output

```
ADMA5017I: Uninstallation of myapp started.
ADMA5104I: Server index entry for myCellManager was updated successfully.
ADMA5102I: Deletion of config data for myapp from config
repository completed successfully.
ADMA5011I: Cleanup of temp dir for app myapp done.
ADMA5106I: Application myapp uninstalled successfully.
```

Examples

- Using Jacl:

```
$AdminApp uninstall myApp
```

- Using Jython:

```
print AdminApp.uninstall('myApp')
```

update

Use the **update** command to update an application in non-interactive mode. This command supports the addition, removal, and update of application subcomponents or the entire application. Provide the application name, content type, and update options.

Target object

None.

Required parameters

application name
Specifies the name of the application to update.

content type
Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list includes the valid content type values for the **update** command:

- `app` - Indicates that you want to update the entire application. This option is the same as indicating the update option with the **install** command. With the `app` value as the content type, you must specify the operation option with update as the value. Provide the new enterprise archive file (EAR) file using the contents option. You can also specify binding information and application options. By

default, binding information for installed modules is merged with the binding information for updated modules. To change this default behavior, specify the update.ignore.old or the update.ignore.new options.

- `file` - Indicates that you want to update a single file. You can add, remove, or update individual files at any scope within the deployed application. With the `file` value as the content type, you must perform operations on the file using the `operation` option. Depending on the type of operation, additional options are required. For file additions and updates, you must provide file content and the file URI relative to the root of the EAR file using the contents and contenturi options. For file deletion, you must provide the file URI relative to the root of the EAR file using the contenturi option which is the only required input. Any other options that you provide are ignored.
- `modulefile` - Indicates that you want to update a module. You can add, remove, or update an individual application module. If you specify the `modulefile` value as the content type, you must indicate the operation that you want to perform on the module using the operation option. Depending on the type of operation, further options are required. For installing new modules or updating existing modules in an application, you must indicate the file content and the file URI relative to the root of the EAR file using the contents and contenturi options. You can also specify binding information and application options that pertain to the new or updated modules. For module updates, the binding information for the installed module is merged with the binding information for the input module by default. To change the default behavior, specify the update.ignore.old or the update.ignore.new options. To delete a module, indicate the file URI relative to the root of the EAR file.
- `partialapp` - Indicates that you want to update a partial application. Using a subset of application components provided in a zip file format you can update, add, and delete files and modules. The zip file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, it contains application artifacts in the same hierarchical structure as they display in an EAR file. For more information on how to construct the partial application zip file, see the Java API section. If you indicate the `partialapp` value as the content type, use the contents option to specify the location of the zip file. When a partial application is provided as an update input, binding information and application options cannot be specified and are ignored, if provided.

Optional parameters

options

There are many options available for the **update** command. For a list of each valid option for the **update** command, see “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 1269.

Sample output

```
Update of singleFile has started.
ADMA5009I: Application archive extracted at C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext
Added files from partial ear: []
performFileOperation: source=C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext,
dest=C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\mrg, uri= META-INF/web.xml, op= add
Copying file from C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext/META-INF/web.xml to
C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\mrg\META-INF\web.xml
Collapse list is: []
FileMergeTask completed successfully
ADMA5005I: Application singleFile configured in WebSphere repository
delFiles: []
delM: null
addM: null
Pattern for remove loose and mod:
Loose add pattern: META-INF/[^/]*|WEB-INF/[^/]*|.*wsdl
root file to be copied: META-INF/web.xml to
C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a191b4e\workspace\cells\BAMBIE\applications\
singleFile.ear\deployments\singleFile/META-INF/web.xml
ADMA5005I: Application singleFile configured in WebSphere repository xmlDoc: [#document: null]
root element: [app-delta: null]
****** delta file name: C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a191b4e\workspace\cells\BAMBIE\applications\
singleFile.ear/deltas/delta-1079548405564
ADMA5005I: Application singleFile configured in WebSphere repository
ADMA6011I: Deleting directory tree C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0
ADMA5011I: Cleanup of temp dir for app singleFile done.
Update of singleFile has ended.
```

Examples

- Using Jacl:
- Using Jython:
- Using Jython list:

updateAccessIDs

Use the **updateAccessIDs** command to update the access ID information for users and groups that are assigned to various roles that are defined in the application. The system reads the access IDs from the user registry and saves the IDs in the application bindings. This operation improves runtime performance of the application. Use this command after installing an application or after editing security role-specific information for an installed application. This method cannot be invoked when the -conntype option for the wsadmin tool is set to `NONE`. You must be connected to a server to invoke this command.

Target object

None.

Required parameters

application name
Specifies the name of the application of interest.

Optional parameters

bALL
The bALL boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` to retrieve access IDs for users or groups that do not have an access ID in the application bindings.

Sample output

```
ADMA5017I: Uninstallation of myapp started.
ADMA5104I: Server index entry for myCellManager was updated successfully.
ADMA5102I: Deletion of config data for myapp from config repository completed successfully.
ADMA5011I: Cleanup of temp dir for app myapp done.
ADMA5106I: Application myapp uninstalled successfully.
```

Examples

- Using Jacl:

```
$AdminApp updateAccessIDs myapp true
```

- Using Jython:

```
print AdminApp.updateAccessIDs('myapp', 'true')
```

updateInteractive

Use the **updateInteractive** command to add, remove, and update application subcomponents or an entire application. When you update an application module or an entire application using interactive mode, the steps that you use to configure binding information are similar to those that apply to the **installInteractive** command. If you update a file or a partial application, the steps that you use to configure the binding information are not available. In this case, the steps are the same as the ones you use with the **update** command.

Target object

None.

Required parameters

application name
Specifies the name of the application to update.

content type
Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list includes the valid content type values for the **updateInteractive** command:

- `app` - Indicates that you want to update the entire application. This option is the same as indicating the update option with the **install** command. With the `app` value as the content type, you must specify the operation option with update as the value. Provide the new enterprise archive file (EAR) file using the contents option. You can also specify binding information and application options. By default, binding information for installed modules is merged with the binding information for updated modules. To change this default behavior, specify the update.ignore.old or the update.ignore.new options.
- `file` - Indicates that you want to update a single file. You can add, remove, or update individual files at any scope within the deployed application. With the `file` value as the content type, you must perform operations on the file using the `operation` option. Depending on the type of operation, additional options are required. For file additions and updates, you must provide file content and the file URI relative to the root of the EAR file using the contents and contenturi options. For file deletion, you must provide the file URI relative to the root of the EAR file using the contenturi option which is the only required input. Any other options that you provide are ignored.
- `modulefile` - Indicates that you want to update a module. You can add, remove, or update an individual application module. If you specify the `modulefile` value as the content type, you must indicate the operation that you want to perform on the module using the operation option. Depending on the type of operation, further options are required. For installing new modules or updating existing modules in an application, you must indicate the file content and the file URI relative to the root of the EAR file using the contents and contenturi options. You can also specify binding information and application options that pertain to the new or updated modules. For module updates, the binding information for the installed module is merged with the binding information for the input module by default. To change the default behavior, specify the update.ignore.old or the update.ignore.new options. To delete a module, indicate the file URI relative to the root of the EAR file.
- `partialapp` - Indicates that you want to update a partial application. Using a subset of application components provided in a zip file format you can update, add, and delete files and modules. The zip file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, it contains application artifacts in the same hierarchical structure as they display in an EAR file. For more information on how to construct the partial application zip file, see the Java API section. If you indicate the `partialapp` value as the content type, use the contents option to specify the location of the zip file. When a partial application is provided as an update input, binding information and application options cannot be specified and are ignored, if provided.

Optional parameters

options
There are many options available for the **updateInteractive** command. For a list of each valid option for the **updateInteractive** command, see “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 1269.

Sample output

```
Getting tasks for: myApp
WASX7266I: A was.policy file exists for this application; would you like to display it? [No]

Task[4]: Binding enterprise beans to JNDI names
Each non message driven enterprise bean in your application or module must be bound to a JNDI name.
EJB Module:  Increment EJB module
EJB:  Increment
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [Inc]:

Task[10]: Specifying the default data source for
```

```
EJB 2.x modules
Specify the default data source for
the EJB 2.x Module containing 2.x CMP beans.

WASX7349I: Possible value for resource authorization is container or per connection factory
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [DefaultDatasource]:
Resource Authorization:  [Per connection factory]:

Task[12]: Specifying data sources for individual 2.x CMP beans
Specify an optional data source for each 2.x CMP bean. Mapping a specific data source to a CMP bean overrides
the default data source for the module containing the enterprise bean.

WASX7349I: Possible value for resource authorization is container or per connection factory
EJB Module:  Increment EJB module
EJB:  Increment
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [DefaultDatasource]:
Resource Authorization:  [Per connection factory]:container
Setting "Resource Authorization" to "cmpBinding.container"

Task[14]: Selecting Application Servers
Specify the application server where you want to install modules that are contained in your application.
Modules can be installed on the same server or dispersed among several servers.
Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
Server:  [WebSphere:cell=myCell,node=myNode,server=server1]:

Task[16]: Selecting method protections for unprotected methods for 2.x EJB
Specify whether you want to assign security role to the unprotected method, add the method to the exclude list, or mark
the method as unchecked.
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
Protection Type:  [methodProtection.uncheck]:
Task[18]: Selecting backend ID
Specify the selection for the BackendID
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
BackendId list:  CLOUDSCAPE_V50_1
CurrentBackendId:  [CLOUDSCAPE_V50_1]:

Task[21]: Specifying application options
Specify the various options available to prepare and install your application.
Pre-compile JSP:  [No]:
Deploy EJBs:  [No]:
Deploy WebServices:  [No]:

Task[22]: Specifying EJB deploy options
Specify the options to deploy EJB.
....EJB Deploy option is not enabled.

Task[24]: Copy WSDL files
Copy WSDL files
....This task does not require any user input

Task[25]: Specify options to deploy Web services
Specify options to deploy Web services
....Web Services deploy option is not enabled.
Update of myApp has started.

ADMA5009I: Application archive extracted at C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a48e969\ext/Increment.jar
FileMergeTask completed successfully
ADMA5005I: Application myApp configured in WebSphere repository
delFiles: []
delM: null
addM: [Increment.jar, ]
Pattern for remove loose and mod:
Loose add pattern: META-INF/[^/]*|WEB-INF/[^/]*|.*wsdl
root file to be copied:
META-INF/application.xml to C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/META-INF/application.xml
del files for full module add/update: []
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ejb-jar.xml
ADMA6016I: Add to workspace Increment.jar/META-INF/ejb-jar.xml
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/MANIFEST.MF
ADMA6016I: Add to workspace Increment.jar/META-INF/MANIFEST.MF
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ibm-ejb-jar-bnd.xmi
ADMA6016I: Add to workspace Increment.jar/META-INF/ibm-ejb-jar-bnd.xmi
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/Table.ddl
ADMA6016I: Add to workspace Increment.jar/META-INF/Table.ddl
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ibm-ejb-jar-ext.xmi
ADMA6016I: Add to workspace Increment.jar/META-INF/ibm-ejb-jar-ext.xmi
add files for full module add/update: [Increment.jar/META-INF/ejb-jar.xml, Increment.jar/META-INF/MANIFEST.MF,
Increment.jar/META-INF/ibm-ejb-jar-bnd.xmi,
```

```
Increment.jar/META-INF/Table.ddl, Increment.jar/META-INF/ibm-ejb-jar-ext.xmi]
ADMA5005I: Application myApp configured in WebSphere repository
xmlDoc: [#document: null]
root element: [app-delta: null]
****** delta file name: C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear/deltas/delta-1079551520393
ADMA5005I: Application myApp configured in WebSphere repository
ADMA6011I: Deleting directory tree C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a48e969
ADMA5011I: Cleanup of temp dir for app myApp done.
Update of myApp has ended.
```

Examples

- Using Jacl:
- Using Jython:
- Using Jython list:

view

Use the **view** command to view the task that is specified by the task name parameter for the application or module that is specified by the application name parameter. Use -tasknames as the option to get a list of valid task names for the application. Otherwise, specify one or more task names as the option.

Target object

None.

Required parameters

name
Specifies the name of the application or module to view.

Optional parameters

bALL
The bALL boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` to retrieve access IDs for users or groups that do not have an access ID in the application bindings.

-buildVersion
Specifies whether to display the build version of the application of interest.

Sample output

The command returns the following information if you specify the `taskoptions` value for the task name parameter:

```
MapModulesToServers
MapWebModToVH
MapRolesToUsers
```

The command returns the following information if you specify the `mapModulesToServers` task for the task name parameter:

```
MapModulesToServers: Selecting Application Servers

Specify the application server where you want to install the modules that are contained in your
application. Modules can be installed on the same server or dispersed among several servers:

Module:  adminconsole
URI:  adminconsole.war,WEB-INF/web.xml
Server:  WebSphere:cell=juniartiNetwork,
node=juniartiManager,server=dmgr
```

Examples

The following **view** command example lists each available task name:

- Using Jacl:

```
$AdminApp view DefaultApplication {-tasknames}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', ['-tasknames'])
```

The following **view** command example returns information for the mapModulesToServer task:

- Using Jacl:

```
$AdminApp view DefaultApplication {-MapModulesToServers}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', ['-MapModulesToServers'])
```

The following **view** command example returns information for the AppDeploymentOptions task:

- Using Jacl:

```
$AdminApp view DefaultApplication {-AppDeploymentOptions}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', '-AppDeploymentOptions')
```

The following **view** command example returns the build version for the DefaultApplication application:

- Using Jacl:

```
$AdminApp view DefaultApplication {-buildVersion}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', '-buildVersion')
```

Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands

This article lists the available options for the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands of the AdminApp object.

You can use the commands for the AdminApp object to install, edit, update, and manage your application configurations. This topic provides additional options to use with the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands to administer your applications. The options listed in this topic apply to all of these commands except where noted.

You can use pattern matching to simplify the task of supplying required values for certain complex options. Pattern matching only applies to fields that are required or read only.

Note: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

The following options are available for the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands:

ActSpecJNDI

The ActSpecJNDI option binds Java 2 Connector (J2C) activation specifications to destination Java Naming and Directory Interface (JNDI) names. You can optionally bind J2C activation specifications in your application or module to a destination JNDI name. You can assign a value to each of the following elements of the ActSpecJNDI option: RARModule, uri, j2cid, and j2c.jndiName fields. The contents of the option after running default bindings include:

- RARModule: <rar module name>
- uri: <rar name>,META-INF/ra.xml
- Object identifier: <messageListenerType>
- JNDI name: null

To use this option, you must specify the `Destination` property in the `ra.xml` file and set the introspected type of the `Destination` property as `javax.jms.Destination`

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $embeddedEar {-ActSpecJNDI {{"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml
 javax.jms.MessageListener jndi5} {"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml javax.jms.MessageListener2 jndi6}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $embeddedEar {-ActSpecJNDI {{.*  .*.rar,.* javax.jms.MessageListener jndi5} {.*
 .*.rar,.*  javax.jms.MessageListener2 jndi6}}}
```

Using Jython:

```
AdminApp.install(embeddedEar, ['-ActSpecJNDI', [["FVT Resource Adapter",
 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageListener', 'jndi5'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml',
 'javax.jms.MessageListener2', 'jndi6']]])
```

Using Jython with pattern matching:

```
AdminApp.install(embeddedEar, ['-ActSpecJNDI', [['.*', '.*.rar,.*', 'javax.jms.MessageListener',
 'jndi5'], ['.*', '.*.rar,.*', 'javax.jms.MessageListener2', 'jndi6']]])
```

allowDispatchRemoteInclude

The allowDispatchRemoteInclude option enables an enterprise application to dispatch includes to resources across web modules that are in different Java virtual machines in a managed node environment through the standard request dispatcher mechanism.

Batch mode example usage

Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:RRDEnabledAppname/] set deploymentObject
 [$AdminConfig showAttribute $deployments deployedObject] set rrdAttr [list allowDispatchRemoteInclude true] set attrs [list
 $rrdLocalAttr] $AdminConfig modify $deploymentObject $attrs
```

allowPermInFilterPolicy

The allowPermInFilterPolicy option specifies that the application server should continue with the application deployment process even when the application contains policy permissions that are in the filter policy. This option does not require a value.

allowServiceRemoteInclude

an enterprise application to service an include request from an enterprise application with the `allowDispatchRemoteInclude` option set to `true`.

Batch mode example usage

Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:RRDEnabledAppname/] set deploymentObject
 [$AdminConfig showAttribute $deployments deployedObject] set rrdAttr [list allowServiceRemoteInclude true] set attrs [list
 $rrdAttr] $AdminConfig modify $deploymentObject $attrs
```

appname

The appname option specifies the name of the application. The default value is the display name of the application.

BackendIdSelection

The BackendIdSelection option specifies the backend ID for the enterprise bean Java archive (JAR) modules that have container-managed persistence (CMP) beans. An enterprise bean JAR module can support multiple backend configurations as specified using an application assembly tool. Use this option to change the backend ID during installation.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-BackendIdSelection
 {{Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml  DB2UDBNT_V72_1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-BackendIdSelection
 {{.* Annuity20EJB.jar,.* DB2UDBNT_V72_1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-BackendIdSelection
 [[Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml  DB2UDBNT_V72_1]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-BackendIdSelection',[['.*', 'Annuity20EJB.jar,.*', 'DB2UDBNT_V72_1']]])
```

BindJndiForEJBBusiness

The BindJndiForEJBBusiness option binds EJB modules with business interfaces to JNDI names. Ensure that each EJB module with business interfaces is bound to a JNDI name.

The current contents of the option after running default bindings include:

- EJBModule: SampleModule
- EJB: SampleEJB
- URI: sample.jar,META-INF/ejb-jar.xml
- Business interface: com.ibm.sample.business.bnd.LocalTargetOne
- JNDI name: []: ejblocal:ejb/LocalTargetOne

If you specify the target resource JNDI name using the BindJndiForEJBNonMessageBinding option, do not specify a business interface JNDI name in the BindJndiForEJBBusiness option. If you do not specify the target resource JNDI name, specify a business interface JNDI name. If you do not specify a business interface JNDI name, the runtime provides a container default.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{SampleModule SampleEJB
 sample.jar,META-INF/ejb-jar.xml com.ibm.sample.business.bnd.LocalTargetOne ejblocal:ejb/LocalTargetOne}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{.*  .*  .*.jar,.*
 com.ibm.sample.business.bnd.LocalTargetOne ejblocal:ejb/LocalTargetOne}}}
```

Using Jython:

```
AdminApp.install(ear, ['-BindJndiForEJBMessageBinding', [['SampleModule', 'SampleEJB',
 'sample.jar,META-INF/ejb-jar.xml', 'com.ibm.sample.business.bnd.LocalTargetOne', 'ejblocal:ejb/LocalTargetOne']]])
```

Using Jython with pattern matching:

```
AdminApp.install(ear, ['-BindJndiForEJBMessageBinding', [['.*', '.*', '.*.jar,.*',
 'com.ibm.sample.business.bnd.LocalTargetOne', 'ejblocal:ejb/LocalTargetOne']]])
```

BindJndiForEJBMessageBinding

The BindJndiForEJBMessageBinding option binds enterprise beans to listener port names or Java Naming and Directory Interface (JNDI) names. Use this option to provide missing data or update a task. Ensure each message-driven enterprise bean in your application or module is bound to a listener port name.

Each element of the BindJndiForEJBMessageBinding option consists of the following fields: EJBModule, EJB, uri, listenerPort, JNDI, jndi.dest, and actspec.auth. Some of these fields, can be assigned values: listenerPort, JNDI, jndi.dest, and actspec.auth.

The current contents of the option after running default bindings include:

- EJBModule: Ejb1
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Listener port: [null]:
- JNDI name: [eis/MessageBean]:
- Destination JNDI Name: [jms/TopicName]:
- ActivationSpec Authentication Alias: [null]:

The default Destination JNDI Name is collected from the corresponding message reference.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{Ejb1 MessageBean
 ejb-jar-ic.jar,META-INF/ejb-jar.xml myListenerPort jndi1 jndiDest1 actSpecAuth1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{.*  .*  .*.jar,.*  myListenerPort jndi1
 jndiDest1 actSpecAuth1}}}
```

Using Jython:

```
AdminApp.install(ear, ['-BindJndiForEJBMessageBinding', [['Ejb1', 'MessageBean',
 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'myListenerPort', 'jndi1', 'jndiDest1', 'actSpecAuth1']]])
```

Using Jython with pattern matching:

```
AdminApp.install(ear, ['-BindJndiForEJBMessageBinding', [['.*', '.*', '.*.jar,.*',
 'myListenerPort', 'jndi1', 'jndiDest1', 'actSpecAuth1']]])
```

BindJndiForEJBNonMessageBinding

The BindJndiForEJBNonMessageBinding option binds enterprise beans to Java Naming and Directory Interface (JNDI) names. Ensure each non message-driven enterprise bean in your application or module is bound to a JNDI name. Use this option to provide missing data or update a task.

The current contents of the option after running default bindings include:

- EJBModule: Ejb1
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Target Resource JNDI Name: [com.ibm.wssvt.acme.annuity.common.business.ejbws.ejb2xjaxrpc.AnnuityMgmtSvcEJB2xJAXRPC]:
- Local Home JNDI Name: [null]:
- Remote Home JNDI Name: [null]:

Special constraints exist for Enterprise JavaBeans (EJB) 3.0 modules. If you specify the target resource JNDI name, do not specify the local home or remote home JNDI names. You also cannot specify the JNDI for business interfaces field in the BindJndiForEJBBusiness option. If you do not specify the target resource JNDI name, then the local and remote home JNDI name fields are optional. If you do not specify local and remote JNDI names, the runtime provides a container default.

If you do not use EJB 3.0 modules, you must specify the target resource JNDI name.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install C:\myapp.ear
 {-BindJndiForEJBNonMessageBinding{{.*  .*  ejb-jar-ic.jar,.*
 com.ibm.wssvt.acme.annuity.common.business.ejbws.ejb2xjaxrpc.AnnuityMgmtSvcEJB2xJAXRPC "" "" }}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-BindJndiForEJBNonMessageBinding {{.*  .*  ejb-jar-ic.jar,.*
 com.ibm.wssvt.acme.annuity.common.business.ejbws.ejb2xjaxrpc.AnnuityMgmtSvcEJB2xJAXRPC "" ""}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-BindJndiForEJBNonMessageBinding [[Ejb1 MessageBean ejb-jar-ic.jar,META-INF/ejb-jar.xml
 com.ibm.wssvt.acme.annuity.common.business.ejbws.ejb2xjaxrpc.AnnuityMgmtSvcEJB2xJAXRPC "" ""]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-BindJndiForEJBNonMessageBinding', [['.*', '.*', 'ejb-jar-ic.jar,.*',
 'com.ibm.wssvt.acme.annuity.common.business.ejbws.ejb2xjaxrpc.AnnuityMgmtSvcEJB2xJAXRPC', '', '']]])
```

blaname

Use the blaname option to specify the name of business level application under which the system creates the Java EE application. This option is optional. If you do not specify a value, the system sets the name as the Java EE application name. This option is available only with the install command.

buildVersion

The buildVersion option displays the build version of an application EAR file. You cannot modify this option because it is read-only. This option returns the build version information for an application EAR if you have specified the build version in the `MANIFEST.MF` application EAR file.

cell

The cell option specifies the cell name to install or update an entire application, or to update an application in order to add a new module. If you want to update an entire application, this option only applies if the application contains a new module that does not exist in the installed application.

Batch mode example usage

Using Jython:

```
AdminApp.install('/myapp/myapp.ear', '[-cell
 cellName]')
```

Using Jacl:

```
$AdminApp install "myapp.ear" {-cell
 cellName}
```

cluster

The cluster option specifies the cluster name to install, or update an entire application or to update an application in order to add a new module. This option only applies in a Network Deployment environment. If you want to update an entire application, this option only applies if the application contains a new module that does not exist in the installed application. You cannot use the -cluster and -server options together.

If you want to deploy an application and specify the HTTP server during the deployment so that the application will appear in the generated `plugin-cfg.xml` file, you must first install the application with a target of -cluster. After you install the application and before you save, use the **edit** command of the AdminApp object to add the additional mapping to the Web server.

Batch mode example usage

Using Jython:

```
AdminApp.install('/myapp/myapp.ear', '[-cluster
 clusterName]')
```

Using Jacl:

```
$AdminApp install "myapp.ear" {-cluster
 clusterName}
```

contents

The contents option specifies the file that contains the content that you want to update. For example, depending on the content type, the file could be an EAR file, a module, a partial zip, or a single file. The path to the file must be local to the scripting client. The `contents` option is required unless you have specified the `delete` option.

contenturi

The contenturi option specifies the URI of the file that you are adding, updating, or removing from an application. This option only applies to the **update** command. The `contenturi` option is required if the content type is `file` or `modulefile`. This option is ignored for other content types.

contextroot

The contextroot option specifies the context root that you use when installing a stand-alone Web archive (WAR) file.

CorrectOracleIsolationLevel

The CorrectOracleIsolationLevel option specifies the isolation level for the Oracle type provider. Use this option to provide missing data or to update a task. The last field of each entry specifies the isolation level. Valid isolation level values are `2` or `4`.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You only need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-CorrectOracleIsolationLevel {{AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-CorrectOracleIsolationLevel {{.*  jms/MyQueueConnectionFactory jms/Resource1 2}}
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-CorrectOracleIsolationLevel', [['.*', 'jms/MyQueueConnectionFactory', 'jms/Resource1', 2]]])
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-CorrectOracleIsolationLevel [[AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2]]]')
```

CorrectUseSystemIdentity

The CorrectUseSystemIdentity option replaces RunAs System to RunAs Roles. The enterprise beans that you install contain a RunAs system identity. You can optionally change this identity to a RunAs role. Use this option to provide missing data or update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-CorrectUseSystemIdentity {{Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml getValue() RunAsUser2 user2 password2}
 {Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment() RunAsUser2 user2 password2}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-CorrectUseSystemIdentity {{.*  .*  .*  getValue() RunAsUser2 user2 password2} {.*  .*  .*  Increment() RunAsUser2 user2
 password2}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-CorrectUseSystemIdentity [[Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml getValue() RunAsUser2 user2 password2]
 [Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment() RunAsUser2 user2 password2]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-CorrectUseSystemIdentity', [['.*', '.*', '.*', 'getValue()', 'RunAsUser2', 'user2', 'password2'], ['.*', '.*', '.*',
 'Increment()', 'RunAsUser2', 'user2', 'password2']]])
```

createMBeansForResources

The createMBeansForResources option specifies that MBeans are created for all resources, such as servlets, JavaServer Pages (JSP) files, and enterprise beans, that are defined in an application when the application starts on a deployment target. This option does not require a value. The default setting is the nocreateMBeansForResources option.

CtxRootForWebMod

The CtxRootForWebMod option edits the context root of the Web module. You can edit a context root that is defined in the `application.xml` file using this option. The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- ContextRoot: <context root>

If the Web module is a Servlet 2.5, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -CtxRootForWebMod {{"IVT Application"
 ivt_app.war,web.xml /mycontextroot}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -CtxRootForWebMod {{.* .* /mycontextroot}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-CtxRootForWebMod', [["IVT Application",
 'ivt_app.war,web.xml', '/mycontextroot']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-CtxRootForWebMod', [['.*', '.*',
 '/mycontextroot']]])
```

custom

The custom option specifies a name-value pair using the format `name=value`. Use the `custom` option to pass options to application deployment extensions. See the application deployment extension documentation for available custom options.

CustomActivationPlan

The CustomActivationPlan option specifies runtime components to add or remove from the default runtime components that are used to run the application. Only use this option when the application server can not obtain all necessary runtime components by inspecting the application.

Batch mode example usage

Using Jython:

```
AdminApp.install('myapp.ear','[-CustomActivationPlan [["Increment EJB module"
 Increment.jar,META-INF/ejb-jar.xml WebSphere:specname=WS_ComponentToAdd ""] ["Default Web Application"
 DefaultWebApplication.war,WEB-INF/web.xml "" ""]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear','[-CustomActivationPlan [[.* Increment.jar,META-INF/ejb-jar.xml
 WebSphere:specname=WS_ComponentToAdd ""] [.* DefaultWebApplication.war,WEB-INF/web.xml "" ""]]')
```

DataSourceFor10CMPBeans

The DataSourceFor10CMPBeans option specifies optional data sources for individual 1.x container-managed persistence (CMP) beans. Use this option to provide missing data or to update a task.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Each element of the DataSourceFor10CMPBeans option consists of the following fields: EJBModule, EJB, uri, JNDI, userName, password, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, userName, password, login.config.name, and auth.props.

The current contents of the option after running default bindings include:

- EJBModule: Increment CMP 1.1 EJB
- EJB: IncCMP11
- URI: IncCMP11.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- User name: [null]:
- Password: [null]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias . The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-DataSourceFor10CMPBeans {{"Increment CMP 1.1 EJB" IncCMP11 IncCMP11.jar,META-INF/ejb-jar.xml myJNDI user1 password1 loginName1
 authProps1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-DataSourceFor10CMPBeans {{.*  .*  IncCMP11.jar,.* myJNDI user1 password1 loginName1 authProps1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor10CMPBeans', [["Increment CMP 1.1 EJB", 'IncCMP11', 'IncCMP11.jar,META-INF/ejb-jar.xml', 'myJNDI', 'user1',
 'password1', 'loginName1', 'authProps1']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor10CMPBeans', [['.*', '.*', 'IncCMP11.jar,.*', 'myJNDI', 'user1', 'password1', 'loginName1', 'authProps1']]])
```

DataSourceFor20CMPBeans

The DataSourceFor20CMPBeans option specifies optional data sources for individual 2.x container-managed persistence (CMP) beans. Use this option to provide missing data or to update a task.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Each element of the DataSourceFor20CMPBeans option consists of the following fields: EJBModule, EJB, uri, JNDI, resAuth, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, resAuth, login.config.name, and auth.props.

The current contents of the option after running default bindings includes the following:

- EJBModule: Increment enterprise bean
- EJB: Increment
- URI: Increment.jar,META-INF/ejb-jar.xml
- JNDI name: [null]:
- Resource authorization: [Per application]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias . The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application. You only need to provide data for rows or entries that are missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-DataSourceFor20CMPBeans {{"Increment EJB module" Increment Increment.jar,META-INF/ejb-jar.xml jndi1 container "" ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-DataSourceFor20CMPBeans {{.*  .*  Increment.jar,.*  jndi1 container "" ""}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor20CMPBeans', [["Increment EJB module", 'Increment', 'Increment.jar,META-INF/ejb-jar.xml', 'jndi1', 'container',
 '', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor20CMPBeans', [['.*', '.*', 'Increment.jar,.*', 'jndi1', 'container', '', '']]])
```

DataSourceFor10EJBModules

The DataSourceFor10EJBModules option specifies the default data source for the enterprise bean module that contains 1.x container-managed persistence (CMP) beans. Use this option to provide missing data or update a task.

Each element of the DataSourceFor10EJBModules option consists of the following fields: EJBModule, uri, JNDI, userName, password, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, userName, password, login.config.name, and auth.props.

The current contents of the option after running default bindings include:

- EJBModule: Increment CMP 1.1 enterprise bean
- uri: IncCMP11.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- User name: [null]:
- Password: [null]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+).

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-DataSourceFor10EJBModules {{"Increment CMP 1.1 EJB" IncCMP11.jar,META-INF/ejb-jar.xml yourJNDI user2 password2 loginName
 authProps}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-DataSourceFor10EJBModules {{.*  IncCMP11.jar,.*  yourJNDI user2 password2 loginName authProps}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor10EJBModules', [["Increment CMP 1.1 EJB", 'IncCMP11.jar,META-INF/ejb-jar.xml', 'yourJNDI', 'user2', 'password2',
 'loginName', 'authProps']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor10EJBModules', [['.*', 'IncCMP11.jar,.*', 'yourJNDI', 'user2', 'password2', 'loginName', 'authProps']]])
```

DataSourceFor20EJBModules

The DataSourceFor20EJBModules option specifies the default data source for the enterprise bean 2.x module that contains 2.x container managed persistence (CMP) beans. Use this option to provide missing data or update a task.

Each element of the DataSourceFor20EJBModules option consists of the following fields: EJBModule, uri, JNDI, resAuth, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, resAuth, login.config.name, auth.props, and extended datasource properties.

The current contents of the option after running default bindings include:

- EJBModule: Increment enterprise bean
- URI: Increment.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- Resource authorization: [Per application]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Extended Data source properties: []: Use this option so that a data source that uses heterogeneous pooling can connect to a DB2 database. The pattern for the property is `property1=value1+property2=value2`.

The last field in each entry of this task specifies the value for resource authorization. Valid values for resource authorization are `per connection factory` or `container`.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias . The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-DataSourceFor20EJBModules {{"Increment EJB module" Increment.jar,META-INF/ejb-jar.xml jndi2 container "" ""
 "clientApplicationInformation=newapplication" ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-DataSourceFor20EJBModules {{.*  Increment.jar,.*  jndi2 container "" "" "clientApplicationInformation=new application"
 ""}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-DataSourceFor20EJBModules', [["Increment EJB module", 'Increment.jar,META-INF/ejb-jar.xml', 'jndi2', 'container', '', '',
 'clientApplicationInformation=newapplication+clientWorkstation=9.10.117.65', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('C:\myapp.ear', ['-DataSourceFor20EJBModules', [['.*',
 'Increment.jar,.*', 'jndi2', 'container', '', '', 'clientApplicationInformation=newapplication+clientWorkstation=9.10.117.65',
 '']]])
AdminApp.install('myapp.ear',
 ['-DataSourceFor20EJBModules', [['.*', 'Increment.jar,.*', 'jndi2', 'container', '', '', 'clientApplicationInformation=new
 application+clientWorkstation=9.10.117.65', '']]])
```

defaultbinding.cf.jndi

The defaultbinding.cf.jndi option specifies the Java Naming and Directory Interface (JNDI) name for the default connection factory.

defaultbinding.cf.resauth

The defaultbinding.cf.resauth option specifies the RESAUTH for the connection factory.

defaultbinding.datasource.jndi

The defaultbinding.datasource.jndi option specifies the Java Naming and Directory Interface (JNDI) name for the default data source.

defaultbinding.datasource.password

The defaultbinding.datasource.password option specifies the password for the default data source.

defaultbinding.datasource.username

The defaultbinding.datasource.username option specifies the user name for the default data source.

defaultbinding.ejbjndi.prefix

The defaultbinding.ejbjndi.prefix option specifies the prefix for the enterprise bean Java Naming and Directory Interface (JNDI) name.

defaultbinding.force

The defaultbinding.force option specifies that the default bindings override the current bindings.

defaultbinding.strategy.file

The defaultbinding.strategy.file option specifies a custom default bindings strategy file.

defaultbinding.virtual.host

The defaultbinding.virtual.host option specifies the default name for a virtual host.

depl.extension.reg (deprecated)

The depl.extension.reg option is deprecated. No replication option is available.

deployejb

The deployejb option specifies to run the EJBDeploy tool during installation. This option does not require a value.

If you pre-deploy the application Enterprise Archive (EAR) file using the EJBDeploy tool then the default value is `nodeployejb`. If not, the default value is `deployejb`.

deployejb.classpath

The deployejb.classpath option specifies an extra class path for the EJBDeploy tool.

deployejb.complianceLevel

The deployejb.complianceLevel option specifies the JDK compliance level for the EJBDeploy tool.

Possible values include:

```
1.4 (default) 5.0 6.0
```

For a list of currently supported JDK compliance levels, run the `ejbdeploy -?` command.

deployejb.dbschema

The deployejb.dbschema option specifies the database schema for the EJBDeploy tool.

deployejb.dbtype

The deployejb.dbtype option specifies the database type for the EJBDeploy tool.

Possible values include:

```
DB2UDB_V81 DB2UDB_V82 DB2UDB_V91 DB2UDB_V95 DB2UDBOS390_V8 DB2UDBOS390_NEWFN_V8 DB2UDBOS390_V9
 DB2UDBISERIES_V53 DB2UDBISERIES_V54 DB2UDBISERIES_V61  DERBY_V10 DERBY_V101 INFORMIX_V100 INFORMIX_V111 INFORMIX_V115
 MSSQLSERVER_2005 ORACLE_V10G ORACLE_V11G SYBASE_V15 SYBASE_V125
```

The following databases support Structured Query Language in Java (SQLJ): DB2UDB_V82, DB2UDB_V81, DB2UDBOS390_V7, and DB2UDBOS390_V8.

For a list of supported database vendor types, run the `ejbdeploy -?` command.

deployejb.dbaccesstype

The deployejb.dbaccesstype option specifies the type of database access. Valid values are `SQLj` and `JDBC`. The default is `JDBC`.

deployejb.rmic

The deployejb.rmic option specifies extra RMIC options to use for the EJBDeploy tool.

deployejb.sqljclasspath

The deployejb.sqljclasspath option specifies the location of the SQLJ translator classes.

deployws

The deployws option specifies to deploy Web services during installation. This option does not require a value.

The default value is: `nodeployws`.

deployws.classpath

The deployws.classpath option specifies the extra class path to use when you deploy Web services.

deployws.jardirs

The deployws.jardirs option specifies the extra extension directories to use when you deploy Web services.

distributeApp

The distributeApp option specifies that the application management component distributes application binaries. This option does not require a value. This setting is the default.

EmbeddedRar

The EmbeddedRar option binds Java 2 Connector objects to JNDI names. You must bind each Java 2 Connector object in your application or module, such as, J2C connection factories, J2C activation specifications and J2C administrative objects, to a JNDI name. Each element of the EmbeddedRar option contains the following fields: `RARModule`, `uri`, `j2cid`, `j2c.name`, `j2c.jndiName`. You can assign the following values to the fields: `j2c.name`, `j2c.jndiName`.

The current contents of the option after running default bindings include:

```
Module: <rar module name> URI: <rar name>,META-INF/ra.xml Object identifier: <identifier
 of the J2C object> name: j2cid JNDI name: eis/j2cid
```

Where j2cid is:

```
J2C connection factory: connectionFactoryInterface  J2C admin object: adminObjectInterface  J2C
 activation specification: message listener type
```

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

If the ID is not unique in the ra.xml file, -<number> will be added. For example, javax.sql.DataSource-2.

Batch mode example usage

Using Jacl:

```
$AdminApp install $embeddedEar {-EmbeddedRar {{"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml
 javax.sql.DataSource javax.sql.DataSource1 eis/javax.sql.javax.sql.DataSSource1} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml javax.sql.DataSource2 javax.sql.DataSource2 eis/javax.sql.DataSource2} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml javax.jms.MessageListener javax.jms.MessageListener1 eis/javax.jms.MessageListener1} {"FVT Resource
 Adapter" jca15cmd.rar,META-INF/ra.xml javax.jms.MessageLListener2 javax.jms.MessageListener2 eis/javax.jms.MessageListener2}
 {"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml fvt.adapter.message.FVTMessageProvider
 fvt.adapter.message.FVTMessageProvider1 eis/fvt.adapter.message.FVTMessageProvider1} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml fvt.adapter.message.FVTMessageProvider2 fvt.adapter.message.FVTMessageProvider2
 eis/fvt.adapter.message.FVTMessageProvider2}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $embeddedEar {-EmbeddedRar {{.*  .*  .*  javax.sql.DataSource1
 eis/javax.sql.javax.sql.DataSSource1} {.*  .*  .*  javax.sql.DataSource2  eis/javax.sql.DataSource2} {.*  .*  .*
 javax.jms.MessageListener1  eis/javax.jms.MessageListener1} {.*  .*  .*  javax.jms.MessageListener2
 eis/javax.jms.MessageListener2} {.*  .*  .*  fvt.adapter.message.FVTMessageProvider1
 eis/fvt.adapter.message.FVTMessageProvider1} {.*  .*  .*  fvt.adapter.message.FVTMessageProvider2
 eis/fvt.adapter.message.FVTMessageProvider2}}}
```

Using Jython:

```
AdminApp.install(embeddedEar, ['-EmbeddedRar', [["FVT Resource Adapter",
 'jca15cmd.rar,META-INF/ra.xml', 'javax.sql.DataSource', 'javax.sql.DataSource1', 'eis/javax.sql.javax.sql.DataSSource1'], ["FVT
 Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml javax.sql.DataSource2', 'javax.sql.DataSource2', 'eis/javax.sql.DataSource2'],
 ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageListener', 'javax.jms.MessageListener1',
 'eis/javax.jms.MessageListener1'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageLListener2',
 'javax.jms.MessageListener2', 'eis/javax.jms.MessageListener2'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml
 fvt.adapter.message.FVTMessageProvider', 'fvt.adapter.message.FVTMessageProvider1',
 'eis/fvt.adapter.message.FVTMessageProvider1'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml',
 'fvt.adapter.message.FVTMessageProvider2', 'fvt.adapter.message.FVTMessageProvider2',
 'eis/fvt.adapter.message.FVTMessageProvider2']]])
```

Using Jython with pattern matching:

```
AdminApp.install(embeddedEar, ['-EmbeddedRar', [['.*', '.*', '.*', 'javax.sql.DataSource1',
 'eis/javax.sql.javax.sql.DataSSource1'], ['.*', '.*', '.*', 'javax.sql.DataSource2', 'eis/javax.sql.DataSource2'], ['.*', '.*',
 '.*', 'javax.jms.MessageListener1', 'eis/javax.jms.MessageListener1'], ['.*', '.*', '.*', 'javax.jms.MessageListener2',
 'eis/javax.jms.MessageListener2'], ['.*', '.*', '.*', 'fvt.adapter.message.FVTMessageProvider1',
 'eis/fvt.adapter.message.FVTMessageProvider1'], ['.*', '.*', '.*', 'fvt.adapter.message.FVTMessageProvider2',
 'eis/fvt.adapter.message.FVTMessageProvider2']]])
```

EnsureMethodProtectionFor10EJB

The EnsureMethodProtectionFor10EJB option selects method protections for unprotected methods of 1.x enterprise beans. Specify to leave the method as unprotected, or assign protection which denies all access. Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-EnsureMethodProtectionFor10EJB {{"Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""} {"Timeout EJB Module"
 TimeoutEJBBean.jar,META-INF/ejb-jar.xml methodProtection.denyAllPermission}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-EnsureMethodProtectionFor10EJB {{.*  IncrementEJBBean.jar,.*  ""} {.*  TimeoutEJBBean.jar,.*
 methodProtection.denyAllPermission}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-EnsureMethodProtectionFor10EJB [["Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""] ["Timeout EJB Module"
 TimeoutEJBBean.jar,META-INF/ejb-jar.xml methodProtection.denyAllPermission]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-EnsureMethodProtectionFor10EJB', [['.*', 'IncrementEJBBean.jar,.*', ""], ['.*', 'TimeoutEJBBean.jar,.*',
 'methodProtection.denyAllPermission']]])
```

The last field in each entry of this task specifies the value of the protection. Valid protection values include: `methodProtection.denyAllPermission`. You can also leave the value blank if you want the method to remain unprotected.

EnsureMethodProtectionFor20EJB

The EnsureMethodProtectionFor20EJB option selects method protections for unprotected methods of 2.x enterprise beans. Specify to assign a security role to the unprotected method, add the method to the exclude list, or mark the method as cleared. You can assign multiple roles for a method by separating roles names with commas. Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require an update the existing data.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear
 {-EnsureMethodProtectionFor20EJB {{CustmerEjbJar customerEjb.jar,META-INF/ejb-jar.xml methodProtection.uncheck} {SupplierEjbJar
 supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-EnsureMethodProtectionFor20EJB {{.*  customerEjb.jar,.*  methodProtection.uncheck} {.*  supplierEjb.jar,.*
 methodProtection.exclude}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-EnsureMethodProtectionFor20EJB [[CustmerEjbJar customerEjb.jar,META-INF/ejb-jar.xml methodProtection.uncheck] [SupplierEjbJar
 supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-EnsureMethodProtectionFor20EJB', [['.*', 'customerEjb.jar,.*', 'methodProtection.uncheck'], ['.*', 'supplierEjb.jar,.*',
 'methodProtection.exclude']]])
```

The last field in each entry of this task specifies the value of the protection. Valid protection values include: `methodProtection.uncheck`, `methodProtection.exclude`, or a list of security roles that are separated by commas.

filepermission

The filepermission option enables you to set the appropriate file permissions on application files that are located in the installation directory. File permissions that you specify at the application level must be a subset of the node level file permission that defines the most lenient file permission that can be specified. Otherwise, node level permission values are used to set file permissions in the installation destination. The file name pattern is a regular expression. The default value is the following:

```
.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755
```

Batch mode example usage

Using Jython:

```
AdminApp.install("/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", ["-appname", "MyApp",
 "-cell", "GooddogNode04Cell", "-node", "GooddogNode", "-server", "server1", "-filepermission",
 ".*\\.jsp=777#.*\\.xml=755"])
```

installdir

The installdir option is deprecated. This option is replaced by the `installed.ear.destination` option.

installed.ear.destination

The installed.ear.destination option specifies the directory to place application binaries.

JSPCompileOptions

The JSPCompileOptions option assigns shared libraries to applications or every module. You can associate multiple shared libraries to applications and modules. The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- JSP Class Path: <jsp class path>
- Use Full Package Names: AppDeploymentOption.Yes | AppDeploymentOption.No
- JDK Source Level: xx
- disableJspRuntimeCompilation.column: AppDeploymentOption.Yes | AppDeploymentOption.No

Specify the options for the JSP precompiler. This option is only valid if you use the preCompileJSPs option also.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -preCompileJSPs -JSPCompileOptions {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml jspcp AppDeploymentOption.Yes 15 AppDeploymentOption.No}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -preCompileJSPs -JSPCompileOptions {{.* .* jspcp
 AppDeploymentOption.Yes 15 AppDeploymentOption.No}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-preCompileJSPs', '-JSPCompileOptions', [["IVT
 Application", 'ivt_app.war,WEB-INF/web.xml', 'jspcp', 'AppDeloymentOption.Yes', 15, 'AppDeploymentOption.No']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', '-appname', 'MyApp', '-preCompileJSPs', '-JSPCompileOptions', [['.*',
 '.*', 'jspcp', 'AppDeploymentOption.Yes', 15, 'AppDeploymentOption.No']]])
```

JSPReloadForWebMod

The JSPReloadForWebMod option edits the JSP reload attributes for the Web module. You can specify the reload attributes of the servlet and JSP for each module. The current contents of the option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- JSP enable Class reloading: <AppDeploymentOption.Yes | AppDeploymentOption.No>
- JSP reload interval in seconds: <jsp reload internal number>

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -JspReloadForWebMod {{"IVT Application"
 ivt_app.war,WEB-INF/ibm-web-ext.xmi AppDeploymentOption.Yes  5}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -JspReloadForWebMod {{.*  .*  AppDeploymentOption.Yes
 5}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-JspReloadForWebMod', [["IVT Application",
 'ivt_app.war,WEB-INF/ibm-web-ext.xmi', 'AppDeploymentOption.Yes', 5]]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-JspReloadForWebMod', [['.*', '.*',
 'AppDeploymentOption.Yes', 5]]])
```

MapEJBRefToEJB

The MapEJBRefToEJB option maps enterprise Java references to enterprise beans. You must map each enterprise bean reference defined in your application to an enterprise bean. Use this option to provide missing data or update to a task.

If the EJB reference is from EJB 3.0, Web 2.4, or Web 2.5 module, the JNDI name is optional. If you specify the useAutoLink option, the JNDI name is optional. Runtime provides a container default. An EJB 3.0 module cannot contain container-managed or bean-managed persistence entity beans. Installation fails when a container-managed or bean-managed persistence entity bean is packaged in a EJB 3.0 module of a Java EE application. You can only package container-managed or bean-managed persistence in an EJB 2.1 or earlier module.

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application. You only need to provide data for rows or entries that are missing information, or those where you want to update the existing data.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapEJBRefToEJB
 {{"Examples Application" "" examples.war,WEB-INF/web.xml BeenThereBean com.ibm.websphere.beenthere.BeenThere IncBean}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapEJBRefToEJB {{.*
 .*  .*  .*  .*  IncBean}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapEJBRefToEJB
 [["Examples Application" "" examples.war,WEB-INF/web.xml BeenThereBean com.ibm.websphere.beenthere.BeenThere IncBean]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapEJBRefToEJB',
 [['.*', '.*', '.*', '.*', '.*', 'IncBean']]])
```

MapEnvEntryForWebMod

The MapEnvEntryForWebMod option edits the env-entry value of the Web module. You can use this option to edit the value of env-entry in the web.xml file.

The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- Name: xxx
- Type: String

- Description: null
- Value: <env-entry value>

If the Web module is a Servlet 2.5, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForWebMod {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml  ivt/ivtEJBObject  String  null  newEnvEntry}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForWebMod {{.*  .*  .*  .* newEnvEntry}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForWebMod', [["IVT Application",
 'ivt_app.war,WEB-INF/web.xml', 'ivt/ivtEJBObject', 'String', 'null', 'newEnvEntry']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForWebMod', [['.*', '.*', '.*',
 '.*', 'newEnvEntry']]])
```

If there is a new line character in the description, use the following syntax:

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForWebMod {{.*  .*  .*
 (?s).* newEnvEntry}}}
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForWebMod', [['.*', '.*', '.*',
 '(?s).*', 'newEnvEntry']]])
```

MapInitParamForServlet

The MapInitParamForServlet option edits the initial parameter of a Web module. You can use this option to edit the initial parameter of a servlet in the web.xml file. The current contents of this option after running the default bindings are the following:

- Web module: xxx
- URI: xxx
- Servlet: xxx
- Name: xxx
- Description: null
- Value: <initial parameter value>

If the Web module is a Servlet 2.5, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -MapInitParamForServlet {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml  ivtservlet  pName1  null  MyInitParamValue}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapInitParamForServlet {{.*  .*  .*  .*  .*
 MyInitParamValue}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapInitParamForServlet', [["IVT Application",
 'ivt_app.war,WEB-INF/web.xml', 'ivtservlet', 'pName1', 'null', 'MyInitParamValue']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapInitParamForServlet', [['.*', '.*', '.*',
 '.*', '.*', 'MyInitParamValue']]])
```

MapMessageDestinationRefToEJB

The MapMessageDestinationRefToEJB option maps message destination references to Java Naming and Directory Interface (JNDI) names of administrative objects from the installed resource adapters. You must map each message destination reference that is defined in your application to an administrative object. Use this option to provide missing data or to update a task.

The current contents of the option after running default bindings include:

- Module: ejb-jar-ic.jar
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Message destination object: jms/GSShippingQueue
- Target Resource JNDI Name: [jms/GSShippingQueue]:

If the message destination reference is from a EJB 3.0 module, then the JNDI name is optional and runtime provides a container default.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $earfile {-MapMessageDestinationRefToEJB {{ejb-jar-ic.jar Publisher
 ejb-jar-ic.jar,META-INF/ejb-jar.xml MyConnection jndi2} {ejb-jar-ic.jar Publisher ejb-jar-ic.jar,META-INF/ejb-jar.xml
 PhysicalTopic jndi3} {ejb-jar-ic.jar Publisher ejb-jar-ic.jar,META-INF/ejb-jar.xml jms/ABC jndi4}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $earfile {-MapMessageDestinationRefToEJB {{.*  .*  .*  MyConnection jndi2} {.*
 .*  .*  PhysicalTopic jndi3} {.*  .*  .*  jms/ABC jndi4}}}
```

Using Jython:

```
AdminApp.install(ear1, ['-MapMessageDestinationRefToEJB', [['ejb-jar-ic.jar', 'Publisher',
 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'MyConnection', 'jndi2'], ['ejb-jar-ic.jar', 'Publisher',
 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'PhysicalTopic', 'jndi3'], ['ejb-jar-ic.jar', 'Publisher',
 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'jms/ABC', 'jndi4']]])
```

Using Jython with pattern matching:

```
AdminApp.install(ear1, ['-MapMessageDestinationRefToEJB', [['.*', '.*', '.*', 'MyConnection',
 'jndi2'], ['.*', '.*', '.*', 'PhysicalTopic', 'jndi3'], ['.*', '.*', '.*', 'jms/ABC', 'jndi4']]])
```

MapModulesToServers

The MapModulesToServers option specifies the application server where you want to install modules that are contained in your application. You can install modules on the same server, or disperse them among several servers. Use this option to provide missing data or to update to a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install /myapp.ear {-MapModulesToServers
 {{"Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml WebSphere:cell=mycell,node=mynode,server=server1} {"Default
 Application" default_app.war,WEB-INF/web.xml WebSphere:cell=mycell,node=mynode,server=server1} {"Examples Application"
 examples.war,WEB-INF/web.xml
 WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=yournode,server=server1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapModulesToServers
 {{.*  .*.jar,.*  WebSphere:cell=mycell,node=mynode,server=server2} {.*  .*.war,.*
 WebSphere:cell=mycell,node=mynode,server=server1}}}
```

The following example, adds *server2* and *server3* to application that is installed:

```
$AdminApp install C:\myapp.ear {-MapModulesToServers {{.* .*
 +WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=mynode,server=server3}} -appname myapp -update
 -update.ignore.old}
```

```
$AdminApp install /myapp.ear {-MapModulesToServers
 {{.* .* +WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=mynode,server=server3}} -appname myapp
 -update -update.ignore.old}
```

The following example removes *server1* from the application that is installed:

```
$AdminApp edit myapp {-MapModulesToServers {{.* .*
 -WebSphere:cell=mycell,node=mynode,server=server1}} -update -update.ignore.old}
```

Using Jython:

```
AdminApp.install('/myapp.ear',
 '[-MapModulesToServers [["Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml
 WebSphere:cell=mycell,node=mynode,server=server1] ["Default Application" default_app.war,WEB-INF/web.xml
 WebSphere:cell=mycell,node=mynode,server=server1] ["Examples Application" examples.war,WEB-INF/web.xml
 WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=yournode,server=server1]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-MapModulesToServers', [['.*', '.*.jar,.*', 'WebSphere:cell=mycell,node=mynode,server=server2']  ['.*', '.*.war,.*',
 'WebSphere:cell=mycell,node=mynode,server=server1']]])
```

The following example, adds *server2* and *server3* to the application that is installed:

```
AdminApp.install('/myapp.ear',
 ['-MapModulesToServers', [['.*', '.*',
 '+WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=mynode,server=server3']], '-appname', 'myapp',
 '-update', '-update.ignore.old'])
```

The following example removes *server1* from the application that is installed:

```
AdminApp.edit('myapp', ['-MapModulesToServers',
 [['.*', '.*', '-WebSphere:cell=mycell,node=mynode,server=server1']]])
```

MapResEnvRefToRes

The MapResEnvRefToRes option maps resource environment references to resources. You must map each resource environment reference that is defined in your application to a resource. Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapResEnvRefToRes
 {{AsyncSender AsyncSender asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapResEnvRefToRes {{.*
  .*  .*  .*  .*  jms/Resource2}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapResEnvRefToRes
 [[AsyncSender AsyncSender asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapResEnvRefToRes',
 [['.*', '.*', '.*', '.*', '.*', 'jms/Resource2']]])
```

MapResRefToEJB

The MapResRefToEJB option maps resource references to resources. You must map each resource reference that is defined in your application to a resource. Use this option to provide missing data or to update a task.

The parameters for MapResRefToEJB include:

- EJBModule: Ejb1
- EJB: MailEJBObject
- URI: deplmtest.jar,META-INF/ejb-jar.xml
- Reference binding: jms/MyConnectionFactory
- Resource type: javax.jms.ConnectionFactory
- JNDI name: [jms/MyConnectionFactory]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Extended Data source properties: []: Use this option so that a data source that uses heterogeneous pooling can connect to a DB2 database. The pattern for the property is property1=value1+property2=value 2.

The DefaultPrincipalMapping login configuration is used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries. If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias . The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is `websphere:name=<name1>,value=<value1>,description=<desc1>`. Specify multiple properties using the plus sign (+).

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapResRefToEJB
 {{deplmtest.jar MailEJBObject deplmtest.jar,META-INF/ejb-jar.xml mail/MailSession9 javax.mail.Session jndi1 login1 authProps1
 "clientApplicationInformation=new application"} {"JavaMail Sample WebApp" "" mtcomps.war,WEB-INF/web.xml mail/MailSession9
 javax.mail.Session jndi2 login2 authProps2 ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapResRefToEJB
 {{deplmtest.jar .*  .*  .*  .*  jndi1 login1 authProps1 "clientApplicationInformation=new application"} {"JavaMail Sample WebApp"
 .*  .*  .*  .*  jndi2 login2 authProps2 ""}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-MapResRefToEJB',
 [['deplmtest.jar', 'MailEJBObject', 'deplmtest.jar,META-INF/ejb-jar.xml mail/MailSession9', 'javax.mail.Session', 'jndi1',
 'login1', 'authProps1', 'clientApplicationInformation=new application+clientWorkstation=9.10.117.65'], ["JavaMail Sample WebApp",
 "", 'mtcomps.war,WEB-INF/web.xml', 'mail/MailSession9', 'javax.mail.Session', 'jndi2', 'login2', 'authProps2', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapResRefToEJB',
 [['deplmtest.jar', '.*', '.*', '.*', '.*', 'jndi1', 'login1', 'authProps1', 'clientApplicationInformation=new
 application+clientWorkstation=9.10.117.65'], ["JavaMail Sample WebApp", '.*', '.*', '.*', '.*', 'jndi2', 'login2', 'authProps2',
 '']]])
```

MapRolesToUsers

The MapRolesToUsers option maps users to roles. You must map each role that is defined in the application or module to a user or group from the domain user registry. You can specify multiple users or groups for a single role by separating them with a pipe (|). Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapRolesToUsers {{"All
 Role" No Yes "" ""} {"Every Role" Yes No "" ""} {DenyAllRole No No user1 group1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapRolesToUsers
 [["All Role" No Yes "" ""]  ["Every Role" Yes No "" ""] [DenyAllRole No No user1 group1]]]')
```

where {{″All Role″ No Yes ″″ ″″} corresponds to the following:

- ″All Role″: Represents the role name
- No: Indicates to allow access to everyone (yes/no)
- Yes: Indicates to allow access to all authenticated users (yes/no)
- ″″: Indicates the mapped users
- ″″: Indicates the mapped groups

MapRunAsRolesToUsers

The MapRunAsRolesToUsers option maps RunAs Roles to users. The enterprise beans that you install contain predefined RunAs roles. Enterprise beans that need to run as a particular role for recognition while interacting with another enterprise bean use RunAs roles. Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapRunAsRolesToUsers
 {{UserRole user1 password1} {AdminRole administrator administrator}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 '[-MapRunAsRolesToUsers [[UserRole user1 password1] [AdminRole administrator administrator]]]')
```

MapSharedLibForMod

The MapSharedLibForMod option assigns shared libraries to application or every module. You can associate multiple shared libraries to applications and modules. The current contents of this option after running default bindings are the following:

- Module: xxx
- URI: META-INF/application.xml
- Shared libraries: <share libraries>

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-appname MyApp -MapSharedLibForMod {{EAR1
 META/application.xml sharedlib1} {"IVT Application" ivt_app.war,WEB-INF/web.xml sharedlib2} {"IVT EJB Module" ivtEJB.jar
 sharedlib3}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapSharedLibForMod {{.*  .*
 sharedlib1} {.*  .*  sharedlib2} {.*  .*  sharedlib3}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapSharedLibForMod', [['EAR1',
 'META/application.xml', 'sharedlib1'], ["IVT Application", 'ivt_app.war,WEB-INF/web.xml', 'sharedlib2'], ["IVT EJB Module",
 'ivtEJB.jar', 'sharedlib3']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapSharedLibForMod', [['.*',
 '.*', 'sharedlib1'], ['.*', '.*', 'sharedlib2'], ['.*', '.*', 'sharedlib3']]])
```

MapWebModToVH

The MapWebModToVH option selects virtual hosts for Web modules. Specify the virtual host where you want to install the Web modules that are contained in your application. You can install Web modules on the same virtual host, or disperse them among several hosts. Use this option to provide missing data or to update a task.

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapWebModToVH
 {{"Default Application" default_app.war,WEB-INF/web.xml default_host} {"Examples Application" examples.war,WEB-INF/web.xml
 default_host}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapWebModToVH {{.*  .*
  default_host}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapWebModToVH
 [["Default Application" default_app.war,WEB-INF/web.xml default_host] ["Examples Application" examples.war,WEB-INF/web.xml
 default_host]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapWebModToVH',
 [['.*', '.*', 'default_host']]])
```

MetadataCompleteForModules

The MetadataCompleteForModules option allows each EJB 3.0 module or Web 2.5 module to write out the complete deployment descriptor including deployment information from annotations. Then the system marks the deployment descriptor for the module as complete. The current contents of this option after running default bindings are the following:

- Module: EJBDD_1.jar
- URI: EJBDD_1.jar,META-INF/ejb-jar.xml
- Lock deployment descriptor: [false]:
- Module: EJBNoDD_2.jar
- URI: EJBNoDD_2.jar,META-INF/ejb-jar.xml
- Lock deployment descriptor: [false]:

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install mayapp.ear
 {-MetadataCompleteForModules {{EJBDD_1.jar EJBDD_1.jar,META-INF/ejb-jar.xml false}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-MetadataCompleteForModules {{.* EJBDD_1.jar,.* false}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-MetadataCompleteForModules', [['EJBDD_1,jar',  'EJBDD_1.jar,META-INF/ejb-jar.xml', 'false']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-MetadataCompleteForModules', [['.*',  'EJBDD_1.jar,.*', 'false']]]
```

noallowDispatchRemoteInclude

The noallowDispatchRemoteInclude option disables the enterprise application that dispatches *includes* to resources across Web modules in different Java virtual machines in a managed node environment through the standard request dispatcher mechanism.

noallowPermInFilterPolicy

The noallowPermInFilterPolicy option specifies not to continue with the application deployment process when the application contains policy permissions that are in the filter policy. This option is the default setting and does not require a value.

noallowServiceRemoteInclude

The noallowServiceRemoteInclude option disables the enterprise application that services an include request from an enterprise application that has the allowDispatchRemoteInclude option set to `true`.

node

The node option specifies the node name to install or update an entire application or to update an application in order to add a new module. If you want to update an entire application, this option only applies if the application contains a new module that does not exist in the installed application.

Batch mode example usage

Using Jython:

```
AdminApp.install('/myapp/myapp.ear', '[-node
 nodeName]')
```

Using Jacl:

```
$AdminApp install "/myapp.ear" {-node
 nodeName}
```

nocreateMBeansForResources

The nocreateMBeansForResources option specifies that MBeans are not created for all resources, such as servlets, JavaServer Pages files, and enterprise beans, that are defined in an application when the application starts on a deployment target. This option is the default setting and it does not require a value.

nodeployejb

The nodeployejb option specifies not to run the EJBDeploy tool during installation. This option is the default setting and does not require a value.

nodeployws

The nodeployws option specifies not to deploy Web services during installation. This option is the default setting and does not require a value.

nodistributeApp

The nodistributeApp option specifies that the application management component does not distribute application binaries. This option does not require a value. The default setting is the distributeApp option.

noreloadEnabled

The noreloadEnabled option disables class reloading. This option does not require a value. The default setting is the `reloadEnabled` option.

nopreCompileJSPs

The nopreCompileJSPs option specifies not to precompile JavaServer Pages files. This option is the default setting and does not require a value.

noprocessEmbeddedConfig

The noprocessEmbeddedConfig option specifies that the system should ignore the embedded configuration data that is include in the application. This option does not required a value. If the application Enterprise Archive (EAR) file does not contain embedded configuration data, the noprocessEmbeddedConfig option is the default setting. Otherwise, the default setting is the processEmbeddedConfig option.

nouseMetaDataFromBinary

The nouseMetaDataFromBinary option specifies that the metadata that is used at run time, for example, deployment descriptors, bindings, extensions, and so on, come from the configuration repository. This option is the default setting and does not require a value. Use this option to indicate that the metadata that is used at run time comes from the enterprise archive file (EAR) file.

nousedefaultbindings

The nousedefaultbindings option specifies not to use default bindings for installation. This option is the default setting and does not require a value.

operation

The operation option specifies the operation that you want to perform. This option only applies to the **update** command. The valid values include:

- `add` - Adds new content.
- `addupdate` - Adds or updates content based on the existence of content in the application.
- `delete` - Deletes content.
- `update` - Updates existing content.

The operation option is required if the content type is file or modulefile. If the value of the content type is `app`, the value of the operation option must be `update`.

Batch mode example usage

The following examples show how to use the options for the **update** command to update a single file in a deployed application:

Using Jacl:

```
$AdminApp update app1 file {-operation update
 -contents /apps/app1/my.xml -contenturi app1.jar/my.xml}
```

Using Jython string:

```
AdminApp.update('app1', 'file', '[-operation update
 -contents /apps/app1/my.xml -contenturi app1.jar/my.xml]')
```

Using Jython list:

```
AdminApp.update('app1', 'file', ['-operation',
 'update', '-contents', '/apps/app1/my.xml', '-contenturi', app1.jar/my.xml'])
```

where `AdminApp` is the scripting object, `update` is the command, `app1` is the name of the application you want to update, `file` is the content type, `operation` is an option of the **update** command, `update` is the value of the `operation`option, `contents` is an option of the **update** command, */apps/app1/my.xml* is the value of the `contents` option, `contenturi` is an option of the **update** command, *app1.jar/my.xml* is the value of the `contenturi` option.

processEmbeddedConfig

The processEmbeddedConfig option processes the embedded configuration data that is included in the application. This option does not required a value. If the application Enterprise Archive (EAR) file contains embedded configuration data, this option is the default setting. If not, the default setting is the nonprocessEmbeddedConfig option.

preCompileJSPs

The preCompileJSPs option specifies to precompile the JavaServer Pages files. This option does not require a value. The default value is nopreCompileJSPs. If you want to precompile JavaServer Pages files, specify it as a part of installation. The default is not to precompile JavaServer Pages files. The precompileJSPs option is ignored during deployment and JavaServer Pages files are not precompiled. The flag is set automatically using assembly tools.

reloadEnabled

The reloadEnabled option specifies that the file system of the application will be scanned for updated files so that changes reload dynamically. This option is the default setting and does not require a value.

reloadInterval

The reloadInterval option specifies the time period in seconds that the file system of the application will be scanned for updated files. Valid range is greater than zero. The default is three seconds.

SharedLibRelationship

The SharedLibRelationship option assigns assets or composition unit IDs as shared libraries for each Java EE module.

The current contents of the option after running default bindings include:

- Module: EJB3BNDBean.jar
- URI: EJB3BNDBean.jar,META-INF/ejb-jar.xml
- Relationship IDs: specify asset or composition unit IDs, such as `[WebSphere:cuname=sharedLibCU1,cuedition=1.0]` or `WebSphere:assetname=sharedLibAsset1.jar`
- Composition Unit names: optionally specify composition unit names for asset relationship IDs. The system uses the same name as the asset if you do not specify a composition unit name. []
- Match target: [Yes]:

You can specify assets and composition unit IDs in the relationship, as the following guidelines explain:

- If you specify an asset, the system creates a composition unit with that asset in the same business level application where the Java EE application belongs.
- If you specify a value for the composition unit names, then the system position matches each name with the corresponding relationship IDs values.
- If the relationship ID is a composition unit ID, then the system ignores the corresponding composition unit name.
- If the relationship ID is an asset ID, then the system creates the composition unit using the corresponding composition unit name.

To specify more than one asset or composition unit ID, separate each value with the plus sign character (+).

When using the **edit** command for the Java EE application, you can override the relationship with a new set of composition unit relationship IDs, or you can add or remove existing composition unit relationships. You cannot specify an asset relationship when using the **edit** command. Specify the first character of the composition unit ID as the plus sign character (+) to add to the relationship, or specify the number sign character (#) to remove the composition unit ID from existing relationships. For example, the `+cuname=cu2.zip` composition unit syntax adds the `cu2` composition unit to the relationship. The `#cuname=cu1.zip+cuname=cu2.zip` composition unit syntax removes the `cu1` and `cu2` composition units from the relationship.

Batch mode example usage

Using Jacl:

```
$AdminApp install mayapp.ear {-SharedLibRelationship
 {{EJB3BNDBean.jar EJB3BNDBean.jar,META-INF/ejb-jar.xml WebSphere:cuname=sharedLibCU1 "" Yes}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-SharedLibRelationship
 {{.* EJB3BNDBean.jar,.* WebSphere:cuname=sharedLibCU1 "" Yes}}}
```

Using Jython:

```
AdminApp.install('myapp.ear',
 ['-SharedLibRelationship', [['EJB3BNDBean,jar',  'EJB3BNDBean.jar,META-INF/ejb-jar.xml', 'WebSphere:cuname=sharedLibCU1,' '',
 'Yes']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-SharedLibRelationship', [['.*',  'EJB3BNDBean.jar,.*', 'WebSphere:cuname=sharedLibCU1', '', 'Yes']]]
```

server

The server option specifies the server name to install or update an entire application or to update an application in order to add a new module. If you want to update an application, this option only applies if the application contains a new module that does not exist in the installed application.

You cannot use the -cluster and -server options together. If you want to deploy an application and specify the HTTP server during the deployment so that the application will appear in the generated plugin-cfg.xml file, you must first install the application with a target of -cluster. After you install the application and before you save, use the **edit** command of the AdminApp object to add the additional mapping to the Web server.

Batch mode example usage

Using Jython:

```
AdminApp.install('/myapp/myapp.ear', '[-server
 serverName]')
```

Using Jacl:

```
$AdminApp install "/myapp.ear" {-server
 serverName}
```

target

The target option specifies the target for the installation functions of the AdminApp object. The following is an example of a target option: `WebSphere:cell= mycell,node=my node,server= myserver`

You can specify multiple targets by delimiting them with a plus (+) sign. By default, the targets that you specify when you install or edit an application replace the existing target definitions in the application. You can use a leading plus (+) or negative (-) sign to add or remove targets without having to specify the targets that are not changed.

Batch mode example usage

Using Jacl:

```
$AdminApp install
 /ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear {-appname MyApp -target
 WebSphere:cell=GooddogCell,node=GooddogNode,server=server2+WebSphere:cell=GooddogCell,node=BaddogNode,server=server3}
```

The following example removes *server3* from the application that is installed:

```
$AdminApp install
 /ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear {-appname MyApp -target
 -WebSphere:cell=GooddogCell,node=BaddogNode,server=server3 -update -update.ignore.old}
```

The following example adds *server4* to the application that is installed:

```
$AdminApp upate app {-appname MyApp -target
 +WebSphere:cell=GooddogCell,node=GooddogNode,server=server4 -contents
 /ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear -operation update -update.ignore.old}
```

Using Jython:

```
AdminApp.install("/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", ["-appname", "MyApp", "-target",
 "WebSphere:cell=GooddogCell,node=GooddogNode,server=server2+WebSphere:cell=GooddogCell,node=BaddogNode,server=server3"])
```

The following example removes *server3* from the application that is installed:

```
AdminApp.install("/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", ["-appname", "MyApp", "-target",
 "-WebSphere:cell=GooddogCell,node=BaddogNode,server=server3", "-update", "-update.ignore.old"])
```

The following example adds *server4* to the application that is installed:

```
AdminApp.upate("app", ["-appname", "MyApp",
 "-target", "+WebSphere:cell=GooddogCell,node=GooddogNode,server=server4", "-contents",
 "/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", "-operation", "update", "-update.ignore.old"])
```

update

The update option updates the installed application with a new version of the enterprise archive file (EAR) file. This option does not require a value.

The application to update, which is specified by the appname option, must already be installed in the WebSphere Application Server configuration. The update action merges bindings from the new version with the bindings from the old version, uninstalls the old version, and installs the new version. The binding information from new version of the EAR file is preferred over the corresponding one from the old version. If any element of binding is missing in the new version, the corresponding element from the old version is used.

update.ignore.new

The update.ignore.new option specifies that during the update action, bindings from the new version of the application are ignored. This option does not require a value. This option applies only if you specify one of the following items:

- The update option for the **install** command.
- The modulefile or app as the content type for the **update** command.

update.ignore.old

The update.ignore.old option specifies that during the update action, the bindings from the installed version of the application are ignored. This option does not require a value. This option applies only if you specify one of the following items:

- The update option for the **install** command.
- The modulefile or app as the content type for the **update** command.

useAutoLink

Use the useAutoLink option to automatically resolve Enterprise Bean (EJB) references from EJB module versions prior to EJB 3.0, and from Web module versions that are prior to version 2.4. If you enable the useAutoLink option, you can optionally specify the JNDI name for MapEJBRefToEJB option. Each module in the application must share one common target to enable autolink support.

Batch mode example usage

Using Jacl:

```
$AdminApp install /myapp.ear {-useAutoLink}
```

Using Jython:

```
AdminApp.install('myapp.ear','[-useAutoLink]')
```

useMetaDataFromBinary

The useMetaDataFromBinary option specifies that the metadata that is used at run time, for example, deployment descriptors, bindings, extensions, and so on, come from the EAR file. This option does not require a value. The default value is `nouseMetaDataFromBinary`, which means that the metadata that is used at run time comes from the configuration repository.

usedefaultbindings

The usedefaultbindings option specifies to use default bindings for installation. This option does not require a value. The default setting is nousedefaultbindings.

To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the wsadmin.properties file in the properties directory of the profile of interest.

validateinstall

The validateinstall option specifies the level of application installation validation. Valid option values include:

- off - Specifies no application deployment validation. This value is the default.
- warn - Performs application deployment validation and continues with the application deployment process even when reported warnings or error messages exist.
- fail - Performs application deployment validation and does not to continue with the application deployment process when reported warnings or error messages exist.

verbose

The verbose option causes additional messages to display during installation. This option does not require a value.

WebServicesClientBindDeployedWSDL

The WebServicesClientBindDeployedWSDL option identifies the client Web service that you are modifying. The scoping fields include: Module, EJB, and Web service. The single mutable value for this task is the deployed WSDL file name. It indicates the Web Services Description Language (WSDL) the client uses.

The Module field identifies the enterprise or Web application within the application. If the module is an enterprise bean , the EJB field identifies a particular enterprise bean within the module. The Web service field identifies the Web service within the enterprise bean or the Web application module. This identifier corresponds to the wsdl:service attribute in the WSDL file, prepended with `service/`, for example, `service/WSLoggerService2`.

The deployed WSDL attribute names a WSDL file relative to the client module. An example of a deployed WSDL for a Web application is the following: `WEB-INF/wsdl/WSLoggerService`.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindDeployedWSDL {{AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindDeployedWSDL {{.*  .*  .*
 META-INF/wsdl/DeployedWsdl1.wsdl}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindDeployedWSDL
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindDeployedWSDL',['.*', '.*', '.*',
 'META-INF/wsdl/DeployedWsdl1.wsdl']]])
```

WebServicesClientBindPortInfo

The WebServicesClientBindPortInfo option identifies the port of a client Web service that you are modifying. The scoping fields include: Module, EJB, Web service and Port. The mutable values for this task include: `Sync Timeout`, `BasicAuth ID`, `BasicAuth Password`, `SSL Config`, and `Overridden Endpoint URI`. The basic authentication and Secure Sockets Layer (SSL) fields affect transport level security, not Web services security.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPortInfo {{AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig
 http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPortInfo {{.*  .*  .*  .*  3000
 newHTTP_ID newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPortInfo [[AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig
 http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPortInfo', [['.*', '.*', '.*',
 '.*', '3000', 'newHTTP_ID', 'newHTTP_pwd', 'sslAliasConfig', 'http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS']]])
```

WebServicesClientBindPreferredPort

The WebServicesClientBindPreferredPort option associates a preferred port (implementation) with a port type (interface) for a client Web service. The immutable values identify a port type of the client Web service that you are modifying. The scoping fields include: Module, EJB, Web service and Port Type. The mutable value for this task is `Port`.

- Port Type - QName (″{namespace} localname″) of a port type that is defined by a wsdl:portType attribute in the WSDL file that identifies an interface.
- Port - QName of a port defined by a wsdl:port attribute within a wsdl:service attribute in a WSDL file that identifies an implementation that has preference.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{.*  .*  .*  .*
 WSLoggerJMSPort}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPreferredPort
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPreferredPort',[['.*', '.*',
 '.*', '.*', 'WSLoggerJMSPort']]])
```

WebServicesServerBindPort

The WebServicesServerBindPort option sets two attributes of a Web service port. The immutable values identify the port of a Web service that you are modifying. The scope fields include: Module, Web service and Port. The mutable values include: `WSDL Service Name`, and `Scope`.

The scope determines the life cycle of implementing the Java bean. The valid values include: `Request` (new instance for each request), `Application` (one instance for each web-app), and `Session` (new instance for each HTTP session).

The scope attribute does not apply to Web services that a Java Message Service (JMS) transport. The scope attribute does not apply to enterprise beans.

The WSDL service name identifies a service when more than one service has the same port name. The WSDL service name is represented as a QName string, for example, `{namespace}localname` .

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesServerBindPort {{AddressBookW2JE.jar
 service/WSLoggerService2 WSLoggerJMS {} Session}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesServerBindPort {{.*  WSClientTestService
 WSClientTest Request} {.*  StockQuoteService StockQuote Application}{.*  StockQuoteService StockQuote2 Session}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesServerBindPort [[AddressBookW2JE.jar
 service/WSLoggerService2 WSLoggerJMS "" Session]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesServerBindPort', [['.*',
 'WSClientTestService', 'WSClientTest', 'Request'], ['.*', 'StockQuoteService', 'StockQuote', 'Application'],['.*',
 'StockQuoteService', 'StockQuote2', 'Session']]])
```

WebServicesClientCustomProperty

The WebServicesClientCustomProperty option supports the configuration of the name value parameter for the description of the client bind file of a Web service. The immutable values identify the port of the Web service that you are modifying. The scope fields include: Module, Web service, and Port. The mutable values include: `name` and `value`.

The format of the `name` and `value` values include a string that represents multiple name and value pairs by using the + character as a separator. For example, name string = ″n1+n2+n3″ value string = ″v1+v2+v3″ yields name/value pairs: {{″n1″ ″v1″}, {″n2″ ″v2″}, {″n3″ ″ v3″}},

Batch mode example usage

Using Jacl:

```
$AdminApp edit WebServicesSamples {-WebServicesClientCustomProperty {{join.jar
 com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest service/StockQuoteService STockQuote propname1
 propValue1}{ejbclientonly.jar Exchange service/STockQuoteService STockQuote propname2 propValue2}}}
```

Using Jacl with pattern matching:

```
$AdminApp edit WebServicesSamples {-WebServicesClientCustomProperty {{join.jar
 com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest  .*  .*  propname1 propValue1}{ejbclientonly.jar Exchange  .*  .*
 propname2 propValue2}}}
```

Using Jython:

```
AdminApp.edit('WebServicesSamples', ['-WebServicesClientCustomProperty', [['join.jar',
 'com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest', 'service/StockQuoteService', 'STockQuote', 'propname1',
 'propValue1'],['ejbclientonly.jar', 'Exchange', 'service/STockQuoteService', 'STockQuote', 'propname2', 'propValue2']]])
```

Using Jython with pattern matching:

```
AdminApp.edit('WebServicesSamples', ['-WebServicesClientCustomProperty', [['join.jar',
 'com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest', '.*', '.*', 'propname1', 'propValue1'],['ejbclientonly.jar', 'Exchange',
 '.*', '.*', 'propname2', 'propValue2']]])
```

WebServicesServerCustomProperty

The WebServicesServerCustomProperty option supports the configuration of the name value parameter for the description of the server bind file of a Web service. The scoping fields include the following: Module, EJB, and Web service. The mutable values for this task include: `name` and `value`.

The format of the these values include a string that represents multiple name and value pairs by using the plus (+) character as a separator. For example, name string = ″n1+n2+n3″ value string = ″v1+v2+v3″ yields name and value pairs: {{″n1″ ″v1″}, {″n2″ ″v2″}, {″n3″ ″ v3″}}.

Batch mode example usage

Using Jacl:

```
$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty {{AddressBookW2JE.jar
 AddressBookService AddressBook com.ibm.websphere.webservices.http.responseContentEncoding deflate}}}
```

Using Jacl with pattern matching:

```
$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty {{.*  .*  .*
 com.ibm.websphere.webservices.http.responseContentEncoding deflate}}}
```

Using Jython:

```
AdminApp.edit ( 'WebServicesSamples', '[ -WebServicesServerCustomProperty [[AddressBookW2JE.jar
 AddressBookService AddressBook com.ibm.websphere.webservices.http.responseContentEncoding deflate]]]')
```

Using Jython with pattern matching:

```
AdminApp.edit ( 'WebServicesSamples', ['-WebServicesServerCustomProperty', [['.*', '.*', '.*',
 'com.ibm.websphere.webservices.http.responseContentEncoding', 'deflate']]])
```

Related reference

“Example: Obtaining option information for AdminApp object commands” on page 1310
Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application.

“Commands for the AdminApp object” on page 1250
Use the AdminApp object to install, modify, and administer applications.

Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands

This table lists all of the options available for the **install**, **installInteractive**, **update**, **updateInteractive**, **edit**, and **editInteractive** commands of the AdminApp object.

The table indicates the applicable commands for each option. Some option names are split on multiple lines for printing purposes.

Option name	install and install Inter active command s - install an applicati on	update and update Inter active command s - Update an applicati on	update and update Inter active command s - Add a module	update and update Inter active command s - Update a module	edit and edit Inter active command s - Edit an applicati on	edit and edit Inter active command s - Edit a module
ActSpecJNDI	Yes	Yes	Yes	Yes	Yes	Yes
`allowDispatch Remote Include`	Yes	Yes	No	No	Yes	No
allowPermInFilterPolicy	Yes	Yes	No	No	No	No

allowService Remote Include	Yes	Yes	No	No	Yes	No
appname	Yes	Yes	No	No	No	No
BackendIdSelection	Yes	Yes	Yes	Yes	No	No
BindJndiForEJBMess ageBinding	Yes	Yes	Yes	Yes	Yes	Yes
BindJndiForEJBNonMess ageBinding	Yes	Yes	Yes	Yes	Yes	Yes
cell	Yes	Yes	Yes	No	No	No
cluster	Yes	Yes	Yes	No	No	No
contents		Yes	Yes	Yes	No	No
contenturi		Yes	Yes	Yes	No	No
contextroot	Yes	Yes	Yes	No	No	No
CorrectOracleIsolation Level	Yes	Yes	Yes	Yes	Yes	Yes
CorrectUseSystem Identity	Yes	Yes	Yes	Yes	Yes	Yes
createMBeansFor Resources	Yes	Yes	No	No	Yes	No
CtxRootForWeb Mod	Yes	Yes	Yes	Yes	Yes	Yes
custom	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10 CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20 CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10 EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20 EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
defaultbinding. datasource. jndi	Yes	Yes	Yes	Yes	No	No
defaultbinding. cf.jndi	Yes	Yes	Yes	Yes	No	No
defaultbinding. cf.resauth	Yes	Yes	Yes	Yes	No	No
defaultbinding. datasource. password	Yes	Yes	Yes	Yes	No	No
defaultbinding. datasource. username	Yes	Yes	Yes	Yes	No	No
defaultbinding. ejbjndi. prefix	Yes	Yes	Yes	Yes	No	No
defaultbinding. force	Yes	Yes	Yes	Yes	No	No
defaultbinding. strategy.file	Yes	Yes	Yes	Yes	No	No

defaultbinding.virtual.host	Yes	Yes	Yes	Yes	No	No
depl.extension.reg (deprecated)	No	No	No	No	No	No
deployejb	Yes	Yes	Yes	Yes	No	No
deployejb.classpath	Yes	Yes	Yes	Yes	No	No
deployejb.complianceLevel	Yes	Yes	Yes	Yes	No	No
deployejb.dbschema	Yes	Yes	Yes	Yes	No	No
deployejb.dbtype	Yes	Yes	Yes	Yes	No	No
deployejb.dbaccesstype	Yes	Yes	Yes	Yes	No	No
deployejb.rmic	Yes	Yes	Yes	Yes	No	No
deployejb.sqljclasspath	Yes	Yes	Yes	Yes	No	No
deployws	Yes	Yes	Yes	Yes	No	No
deployws.classpath	Yes	Yes	Yes	Yes	No	No
deployws.jardirs	Yes	Yes	Yes	Yes	No	No
distributeApp	Yes	Yes	No	No	Yes	No
EmbeddedRar	Yes	Yes	Yes	Yes	Yes	Yes
EnsureMethodProtectionFor10EJB	Yes	Yes	Yes	Yes	No	No
EnsureMethodProtentionFor20EJB	Yes	Yes	Yes	Yes	No	No
filepermission	Yes	Yes	No	No	Yes	No
JSPCompileOptions	Yes	Yes	Yes	Yes	No	No
JSPReloadForWebMod	Yes	Yes	Yes	Yes	Yes	Yes
installdir (deprecated)	No	No	No	No	No	No
installed.ear.destination	Yes	Yes	No	No	Yes	No
MapInitParamForServlet	Yes	Yes	Yes	Yes	Yes	Yes
MapMessageDestinationRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapModulesToServers	Yes	Yes	Yes	Yes	Yes	Yes
MapEJBRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapEnvEntryForWebMod	Yes	Yes	Yes	Yes	Yes	Yes
MapResEnvRefToRes	Yes	Yes	Yes	Yes	Yes	Yes
MapResRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapRolesToUsers	Yes	Yes	No	No	Yes	Yes
MapRunAsRolesToUsers	Yes	Yes	Yes	Yes	Yes	Yes
MapSharedLibForMod	Yes	Yes	Yes	Yes	Yes	Yes
MapWebModToVH	Yes	Yes	Yes	Yes	Yes	Yes

noallowDispatch Remote Include	Yes	Yes	No	No	Yes	No
noallowPermIn FilterPolicy	Yes	Yes	No	No	No	No
noallowService Remote Include	Yes	Yes	No	No	Yes	No
nocreateMBeans ForResources	Yes	Yes	No	No	Yes	No
node	Yes	Yes	Yes	No	No	No
nodeployejb	Yes	Yes	Yes	Yes	No	No
nodeployws	Yes	Yes	Yes	Yes	No	No
nodistributeApp	Yes	Yes	No	No	Yes	No
nopreCompileJSPs	Yes	Yes	Yes	Yes	No	No
noprocessEmbedded Config	Yes	Yes	No	No	No	No
noreloadEnabled	Yes	Yes	No	No	Yes	No
nousedefaultbindings	Yes	Yes	Yes	Yes	No	No
nouseMetaData FromBinary	Yes	Yes	No	No	Yes	No
operation	No	Yes	Yes	Yes	No	No
preCompileJSPs	Yes	Yes	Yes	Yes	No	No
processEmbedded Config	Yes	Yes	No	No	No	No
reloadEnabled	Yes	Yes	No	No	Yes	No
reloadInterval	Yes	Yes	No	No	Yes	No
server	Yes	Yes	Yes	No	No	No
target	Yes	Yes	Yes	No	No	No
update	Yes	Yes	No	No	No	No
update.ignore.old	Yes	Yes	No	Yes	No	No
update.ignore.new	Yes	Yes	No	Yes	No	No
useMetaData FromBinary	Yes	Yes	No	No	Yes	No
usedefaultbindings	Yes	Yes	Yes	Yes	No	No
validateinstall	Yes	No	No	No	Yes	No
verbose	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindingDeployed WSDL	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindPortInfo	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindPreferred Port	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient CustomProperty	Yes	Yes	Yes	Yes	Yes	Yes

WebServicesServer BindPort	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesServer CustomProperty	Yes	Yes	Yes	Yes	Yes	Yes

Example: Obtaining option information for AdminApp object commands

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application.

You need to provide data for rows or entries that are either missing information, or require an update.

- You can use the **options** command to see the requirements for an enterprise archive file (EAR) file if you construct installation command lines. The **taskInfo** command provides detailed information for each task option with a default binding applied to the result.
- The options for the AdminApp **install** command can be complex if you specify various types of binding information, for example, Java Naming and Directory Interface (JNDI) name, data sources for enterprise bean modules, or virtual hosts for Web modules. An easy way to specify command-line installation options is to use a feature of the **installInteractive** command that generates the options for you. After you install the application interactively once and specify all the updates that you need, look for message WASX7278I in the wsadmin output log. The default output log for wsadmin is `wsadmin.traceout`. You can cut and paste the data in this message into a script, and modify it.

Commands for the AdminTask object

Use the AdminTask object to run administrative commands with the wsadmin tool.

Administrative commands are loaded dynamically when you start the wsadmin tool. The administrative commands that are available for you to use, and what you can do with them, depends on the edition of the product that you use.

You can start the scripting client without having a server running by using the `-conntype NONE` option with the wsadmin tool. The AdminTask administrative commands are available in both connected and local modes. If a server is currently running, it is not recommended to run the AdminTask commands in local mode because any configuration changes made in local mode are not reflected in the running server configuration and vice versa. If you save a conflicting configuration, you can corrupt the configuration.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you cannot update the configuration because the configuration for these server processes are copies of the master configuration, which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. To change a configuration, do not run a scripting client in local mode on a node machine because this is not supported.

The following AdminTask commands are available but do not belong to a group:

- “configureTAM” on page 1311
- “createTCPEndPoint” on page 1311
- “getTCPEndPoint” on page 1312
- “help” on page 1313
- “listTAMSettings” on page 1314
- “listTCPEndPoints” on page 1315

configureTAM

Use the configureTAM command to manually configure the Tivoli Access Manager.

Target object

None.

Required parameters

None.

Optional parameters

None.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask configureTAM {-interactive}
```

- Using Jython:

```
AdminTask.configureTAM('-interactive')
```

createTCPEndPoint

The createTCPEndPoint command creates a new endpoint that you can associate with a TCP inbound channel.

Target object

Parent instance of the TransportChannelService that contains the TCPInboundChannel. (ObjectName, required)

Required parameters

-name
Specifies the name for the new endpoint. (String, required)

-host
Specifies the host for the new endpoint. (String, required)

-port
Specifies the port for the new endpoint. (String, required)

Optional parameters

None.

Sample output

The command returns the object name of the endpoint that was created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createTCPEndPoint (cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)
  {-name Sample_End_Pt_Name -host mybuild.location.ibm.com -port 8978}
```

- Using Jython string:

```
AdminTask.createTCPEndPoint('cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1',
  '[-name Sample_End_Pt_Name -host mybuild.location.ibm.com -port 8978]')
```

- Using Jython list:

```
AdminTask.createTCPEndPoint('cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1',
 ['-name', 'Sample_End_Pt_Name', '-host', 'mybuild.location.ibm.com', '-port', '8978'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createTCPEndPoint {-interactive}
```

- Using Jython:

```
AdminTask.createTCPEndPoint('-interactive')
```

getTCPEndPoint

The getTCPEndPoint command obtains the named end point that is associated with either a TCP inbound channel or a chain that contains a TCP inbound channel.

Target object

TCPInboundChannel, or containing chain, instance that is associated with a NamedEndPoint. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the object name of an existing named end point that is associated with the TCP inbound channel instance or a channel chain.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getTCPEndPoint TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)

$AdminTask getTCPEndPoint DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)
```

- Using Jython string:

```
print AdminTask.getTCPEndPoint('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)')
print AdminTask.getTCPEndPoint('DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)')
```

- Using Jython list:

```
print AdminTask.getTCPEndPoint('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)')
print AdminTask.getTCPEndPoint('DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getTCPEndPoint {-interactive}
```

- Using Jython:

```
print AdminTask.getTCPEndPoint('-interactive')
```

help

The **help** command provides a summary of the help commands and ways to invoke an administrative command. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Optional parameters

-commands
The **help** command provides a list of available administrative commands if you use the `-commands` parameter. (String, optional)

-commandGroups
The **help** command provides a list of administrative command groups if you use the `-commandGroups` parameter. (String, optional)

-commandName
The **help** command provides help information for a given administrative command. (String, optional)

-stepName
The **help** command provides help information for a given step of an administrative command. (String, optional)

Sample output

The command returns general command information for the AdminTask object.

Examples

Batch mode example usage:

The following command examples return general help information for the AdminTask object:

- Using Jacl:

```
$AdminTask help
```

- Using Jython:

```
print AdminTask.help()
```

The following command examples return display each command for the AdminTask object:

- Using Jacl:

```
$AdminTask help -commands
```

- Using Jython:

```
print AdminTask.help('-commands')
```

The following command examples return detailed command information for the createJ2CConnectionFactory command for the AdminTask object:

- Using Jacl:

```
$AdminTask help createJ2CConnectionFactory
```

- Using Jython:

```
print AdminTask.help('createJ2CConnectionFactory')
```

The following examples demonstrate the use of the wildcard character (*) to return each command that contains the `create` string:

- Using Jacl:

```
$AdminTask help -commands *create*
```

- Using Jython:

```
print AdminTask.help('-commands *create*')
```

The following examples demonstrate the syntax to use regular Java expressions (.*):

- Using Jacl:

```
$AdminTask help -commands <pattern>
```

- Using Jython:

```
print AdminTask.help('-commands <pattern>')
```

listTAMSettings

The listSSLRepertoires command displays the current embedded Tivoli Access Manager configuration settings.

Target object

None.

Required parameters

None.

Optional parameters

None.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listTAMSettings {-interactive}
```

- Using Jython:

```
print AdminTask.listTAMSettings('-interactive')
```

listTCPEndPoints

The listTCPEndPoints command lists all the named end points that can be associated with a TCP inbound channel.

Target object

TCP Inbound Channel instance for which named end points candidates are listed. (ObjectName, required)

Required parameters

None.

Optional parameters

-excludeDistinguished
Specifies whether to show only non-distinguished named end points. This parameter does not require a value. (Boolean, optional)

-unusedOnly
Specifies whether to show the named end points not in use by other TCP inbound channel instances. This parameter does not require a value. (Boolean, optional)

Sample output

The command returns a list of object names for the eligible named end points.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
  {-excludeDistinguished}
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
   {-excludeDistinguished -unusedOnly}
```

- Using Jython string:

```
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished]')
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished]')
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished -unusedOnly]')
```

- Using Jython list:

```
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
  ['-excludeDistinguished'])
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   ['-excludeDistinguished'])
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
  ['-excludeDistinguished', '-unusedOnly'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listTCPEndPoints {-interactive}
```

- Using Jython:

```
print AdminTask.listTCPEndPoints('-interactive')
```

listTCPThreadPools

The listTCPThreadPools command lists all of the thread pools that can be associated with a TCP inbound channel or TCP outbound channel.

Target object

TCPInboundChannel or TCPOutboundChannel instance for which ThreadPool candidates are listed. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of eligible thread pool object names.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listTCPThreadPools TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
```

- Using Jython string:

```
print AdminTask.listTCPThreadPools('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)')
```

- Using Jython list:

```
print AdminTask.listTCPThreadPools('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listTCPThreadPools {-interactive}
```

- Using Jython:

```
print AdminTask.listTCPThreadPools('-interactive')
```

modifyTAM

The **modifyTAM** command modifies embedded Tivoli Access Manager configuration settings.

Target object

None.

Required parameters

-adminPasswd
Specifies the Tivoli Access Manager administrator password. (String, required)

Optional parameters

-adminUid
Specifies the Tivoli Access Manager user name. (String, optional)

-nodeName
Specifies the target node or nodes. Set the value as the * asterisk character to specify all nodes. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyTAM {-adminPasswd my11password}
```

- Using Jython:

```
AdminTask.modifyTAM('-adminPasswd my11password')
```

- Using Jython list:

```
AdminTask.modifyTAM(['-adminPasswd', 'my11password'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyTAM {-interactive}
```

- Using Jython:

```
AdminTask.modifyTAM('-interactive')
```

reconfigureTAM

The reconfigureTAM command reconfigures the Java Authorization Contract for Containers (JACC) Tivoli Access Manager settings.

Target object

None.

Required parameters

None.

Optional parameters

None.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask reconfigureTAM {-interactive}
```

- Using Jython:

```
AdminTask.reconfigureTAM('-interactive')
```

setResourceProperty

Use the setResourceProperty command to set the value of a specified property defined on a resource provider such as JDBCProvider or a connection factory such as DataSource or JMSConnectionFactory. If the property with specified key is defined already, then this command overrides the value. If no property with a specified key is defined, this command will add the property with specified key and value.

Target object

The configuration object ID of a resource provider or a connection factory.

Required parameters

-propertyName
Specifies the name of the property. (String, required)

-propertyValue
Specifies the value of a property. (String, required)

Optional parameters

-propertyType
Specifies the type of the property. The default value is java.lang.String. (String, optional)

-propertyDescription
Specifies the description of the defined property. (String, optional)

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setResourceProperty {-propertyName test.property -propertyValue testValue}
```

- Using Jython string:

```
AdminTask.setResourceProperty('[-propertyName test.property -propertyValue testValue]')
```

- Using Jython list:

```
AdminTask.setResourceProperty(['-propertyName', 'test.property', '-propertyValue', 'testValue'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setResourceProperty {-interactive}
```

- Using Jython:

```
AdminTask.setResourceProperty('-interactive')
```

showResourceProperties

Use the showResourceProperties command to list all of the property values that are defined on a resource provider such as JDBC provider or a connection factory such as data source or JMS connection factory.

Target object

The configuration object ID of a resource provider or a connection factory.

Required parameters

None.

Optional parameters

-propertyName
Specifies the name of the property. If you specify the property name, the value of the specified property name is returned. If you do not specify the property name, all property values will be listed. Each element in the list is a property name value pair. (String, optional)

Sample output

The command returns the property values that are defined on the resource provider or the connection factory that you specified.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showResourceProperties {-propertyName test.property}
```

- Using Jython string:

```
print AdminTask.showResourceProperties('[-propertyName test.property]')
```

- Using Jython list:

```
print AdminTask.showResourceProperties(['-propertyName', 'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showResourceProperties {-interactive}
```

- Using Jython:

```
print AdminTask.showResourceProperties('-interactive')
```

unconfigureTAM

The unconfigureTAM command removes configuration data for the Java Authorization Contract for Containers (JACC) Tivoli Access Manager.

Required parameters

None.

Optional parameters

None.

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask unconfigureTAM {-interactive}
```

- Using Jython:

```
AdminTask.unconfigureTAM('-interactive')
```

updateAppOnCluster

The updateAppOnCluster command can be used to synchronize nodes and restart cluster members for an application update that is deployed to a cluster. After an application update, this command can be used to synchronize the nodes without stopping all the cluster members on all the nodes at one time. This command synchronizes one node at a time. Each node is synchronized by stopping the cluster members on which the application is targeted, performing a node synchronization operation, and restarting the cluster members.

This command might take more time than the default connector timeout period, depending on the number of nodes that the target cluster spans. Be sure to set proper timeout values in the `soap.client.props` file

in the *profile_root*/properties directory, when a SOAP connector is used; in the sas.client.props file, when a JSR160RMI connecor or an RMI connector is used; and in the ipc.client.props file when an IPC connector is used.

This command is not supported in local mode.

Target object

None.

Required parameters

-ApplicationNames
Specifies the names of the applications that are updated. (String, required)

Optional parameters

-timeout
Specifies the timeout value in seconds for each node synchronization. The default is 300 seconds. (Integer, optional)

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateAppOnCluster {-ApplicationNames app1}
$AdminTask updateAppOnCluster {-ApplicationNames app1 -timeout 600}
```

- Using Jython string:

```
AdminTask.updateAppOnCluster('[-ApplicationNames app1]')
AdminTask.updateAppOnCluster('[-ApplicationNames app1 -timeout 600]')
```

- Using Jython list:

```
AdminTask.updateAppOnCluster(['-ApplicationNames', 'app1'])
AdminTask.updateAppOnCluster(['-ApplicationNames', 'app1', '-timeout', '600'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateAppOnCluster -interactive
```

- Using Jython:

```
AdminTask.updateAppOnCluster('-interactive')
```

Administrative command invocation syntax

The administrative command uses a specific syntax to invoke operations.

You can use an administrative command in batch mode or in interactive mode. The following syntax is used for an administrative command:

Using Jacl:

```
$AdminTask cmdName [targetObject] [options]
```

where options include:

```
{
  [-paramName paramValue] [-paramName] ...
  [-stepName {{stepParamValue ...} ...} ...]
  [-delete {-stepName {{stepKeyParamValue ...} ...} ...} ...]
  [-interactive]
}
```

or

```
{
  [-paramName paramValue] [-paramName] ...
  [-stepName {{stepParamName stepParamValue} {stepParamName stepParamValue} ...]
  [-delete {-stepName {{stepKeyParamValue ...} ...} ...} ...]
  [-interactive]
}
```

Using Jython:

```
AdminTask.cmdName(['targetObject'], [options])
```

where `options` include:

```
'[
[-paramName paramValue] [-paramName ...]
[-stepName [[stepParamValue ...] ...] ...]
[-delete [-collectionStepName [[stepKeyParamValue ...] ...] ...] ...]
[-interactive]
]'
```

or

```
'[
[-paramName paramValue] [-paramName ...]
[-stepName [[stepParamName stepParamValue] [stepParamName stepParamValue] ...]
[-delete [-collectionStepName [[stepKeyParamValue ...] ...] ...] ...]
[-interactive]
]'
```

where:

`cmdName`	represents the name of an administrative command to run.
`targetObject`	represents the target object on which the command operates. Depending on the administrative command, this input can be required, optional, or nonexistent. This input corresponds to the Target object that is displayed in the command-specific help.
`paramName`	represents the parameter name of the command that was run. Depending on the administrative command, this input can be required, optional, or nonexistent. Each parameter name corresponds to an argument name that is displayed in the Arguments area of the command-specific help.
`paramValue`	represents the parameter value to set for the preceding parameter name. Parameters are specified as name-value pairs. The parameter value is not required if a parameter has Boolean as its value type. If you specify the parameter name only, without specifying a value for a Boolean type parameter, the value is set to `true`.
`stepName`	represents the step name of the command. This input corresponds to a step name that is displayed in the Steps area of the command-specific help.
`stepParamName`	represents the parameter name for a step. Depending on the administrative command, this input can be required, optional, or nonexistent. Each parameter name corresponds to an argument name that displays in the step area of the command-specific help.

`stepParamValue ...`	represents the values of the parameters for a step. Provide all the parameter values of a step in the correct order, as displayed in the step-specific help. For any optional parameters that you do not want to specify a value, put `""` instead of the value. If a command step is a collection type, for example, it contains multiple objects where each object has the same set of parameters. You can specify multiple objects with each object enclosed by a pair of braces. For collection type steps, each step parameter is a key or a non-key. Key parameters in a step are used to uniquely identify an object in the collection. If data exists in the step, key parameter values that are provided in the input are compared with key parameter values in the existing data. If a match is found, the existing data is updated. Otherwise, if the specified step supports the addition of new objects, the input values are added.
`delete`	represents the option to delete existing data from a specified step that supports collection.
`collectionStepName`	represents the collection step name.
`stepKeyParamValue ...`	represents the values of key parameters to uniquely identify an object to delete from a collection step. You must provide the key parameter values of an object in the order that they are displayed in the step specific help.
`interactive`	represents the option to enter interactive mode.
`[ ]`	represents a Jython list bracket.
[ ]	indicates that the parameters or options inside the brackets are optional. Do not type these brackets as part of the syntax.

Administrative properties for scripting

Scripting administration utilizes several Java property files. Property files can be used to control your system configurations. Before any property file is specified on the command line, three levels of default property files are loaded. These property files include an installation default file, a user default file, and a properties file.

The first level represents an installation default, located in the `profile_root/properties` directory for each application server profile called `wsadmin.properties`. The second level represents a user default, and is located in the Java user.home property. This properties file is also called `wsadmin.properties`. The third level is a properties file that is pointed to by the `WSADMIN_PROPERTIES` environment variable. This environment variable is defined in the environment where the wsadmin tool starts.

If one or more of these property files is present, they are interpreted before any properties file that is present on the command line. The three levels of property files load in the order that they are specified. The properties file that is loaded last overrides the ones loaded earlier.

The following Java properties are used by scripting:

com.ibm.ws.scripting.appendTrace

Determines if the trace file appends to the end of the existing log file. The default setting, `false`, overrides the log file on each invocation.

com.ibm.ws.scripting.classpath

Searches for classes and resources, and is appended to the list of paths.

com.ibm.ws.scripting.connectionType

Determines the connector to use. This value can either be `SOAP`, `JSR160RMI`, `RMI`, `IPC`, or `NONE`. The `wsadmin.properties` file specifies `SOAP` as the connector.

com.ibm.ws.scripting.crossDocumentValidationEnabled

Determines whether the validation mechanism examines other documents when changes are made to one document.

Possible values are `true` and `false`. The default value is `true`.

com.ibm.ws.scripting.defaultLang

Indicates the language to use when running scripts. The `wsadmin.properties` file specifies Jacl as the scripting language.

The supported scripting languages are Jacl and Jython.

com.ibm.ws.scripting.echoparams

Determines if the parameters or arguments output to STDOUT or to a wsadmin log file. The default setting, `true`, outputs the parameters or arguments to a log file.

com.ibm.ws.scripting.emitWarningForCustomSecurityPolicy

Controls whether the WASX7207W message is emitted when custom permissions are found.

The possible values are `true` and `false`. The default value is `true`.

com.ibm.ws.scripting.host

Determines the host to use when attempting a connection. If not specified, the default is the local machine.

com.ibm.ws.scripting.ipchost

The `ipchost` property determines the host that the system uses to connect to the IPC connector. Use the host name or IP address of the loopback adapter that the IPC connector listens to, such as `localhost`, `127.0.0.1`, or `::1`. The default value is `localhost`.

com.ibm.ws.scripting.port

Specifies the port to use when attempting a connection. The `wsadmin.properties` file specifies `8879` as the SOAP port for a single server installation.

com.ibm.ws.scripting.profiles

Specifies a list of profile scripts to run automatically before running user commands, scripts, or an interactive shell.

The `wsadmin.properties` file specifies `securityProcs.jacl` and `LTPA_LDAPSecurityProcs.jacl` as the values of this property. If Jython is specified with the wsadmin -lang option, the wsadmin tool performs a conversion to change the profile script names that are specified in this property to use the file extension that matches the language specified. Use the provided script procedures with the default settings to make security configuration easier.

com.ibm.ws.scripting.traceFile

Determines where trace and log output is directed. The `wsadmin.properties` file specifies the `wsadmin.traceout` file that is located in the `profile_root/logs` directory of each application server profile as the value of this property.

If multiple users work with the wsadmin tool simultaneously, set different traceFile properties in the user properties files. If the file name contains double-byte character set (DBCS) characters, use a unicode format, such as `\uxxxx`, where `xxxx` is a number.

com.ibm.ws.scripting.traceString

Turns on tracing for the scripting process. The default has tracing turned off.

com.ibm.ws.scripting.tempdir

Determines the directory to use for temporary files when installing applications.

The Java virtual machine (JVM) API uses `java.io.temp` as the default value.

com.ibm.ws.scripting.validationLevel

Determines the level of validation to use when configuration changes are made from the scripting interface.

Possible values are: `NONE`, `LOW`, `MEDIUM`, `HIGH`, `HIGHEST`. The default is `HIGHEST`.

com.ibm.ws.scripting.validationOutput

Determines where the validation reports are directed. The default file is `wsadmin.valout` which is located in the `profile_root/logs` directory of each application server profile.

If multiple users work with the wsadmin tool simultaneously, set different validationOutput properties in the user properties files. If the file name contains double-byte character set (DBCS) characters, use unicode format, such as `\uxxxx`, where `xxxx` is a number.

Related tasks

Chapter 17, “Scripting and command line reference material,” on page 1181
Use this topic to locate wsadmin tool commands for the AdminTask, AdminControl, AdminConfig, and AdminApp scripting objects. This topic also provides a pointer to command line commands and options.

Appendix. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories infer specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Java EE WebSphere Application Client is the /QIBM/ProdData/WebSphere/AppClient/V7/client directory.

app_client_user_data_root
The default Java EE WebSphere Application Client user data root is the /QIBM/UserData/WebSphere/AppClient/V7/client directory.

app_client_profile_root
The default Java EE WebSphere Application Client profile root is the /QIBM/UserData/WebSphere/AppClient/V7/client/profiles/*profile_name* directory.

app_server_root
The default installation root directory for WebSphere Application Server Network Deployment is the /QIBM/ProdData/WebSphere/AppServer/V7/ND directory.

cip_app_server_root
The default installation root directory is the /QIBM/ProdData/WebSphere/AppServer/V7/ND/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

A CIP is a WebSphere Application Server Network Deployment product bundled with optional maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts.

cip_profile_root
The default profile root directory is the /QIBM/UserData/WebSphere/AppServer/V7/ND/cip/*cip_uid*/profiles/*profile_name* directory for a customized installation package (CIP) produced by the Installation Factory.

cip_user_data_root
The default user data root directory is the /QIBM/UserData/WebSphere/AppServer/V7/ND/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

if_root This directory represents the root directory of the IBM WebSphere Installation Factory. Because you can download and unpack the Installation Factory to any directory on the file system to which you have write access, this directory's location varies by user. The Installation Factory is an Eclipse-based tool which creates installation packages for installing WebSphere Application Server in a reliable and repeatable way, tailored to your specific needs.

iip_root
This directory represents the root directory of an *integrated installation package* (IIP) produced by the IBM WebSphere Installation Factory. Because you can create and save an IIP to any directory on the file system to which you have write access, this directory's location varies by user. An IIP is an aggregated installation package created with the Installation Factory that can include one or more generally available installation packages, one or more customized installation packages (CIPs), and other user-specified files and directories.

java_home
The following directories are the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
Classic JVM	/QIBM/ProdData/Java400/jdk6
32–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit
64–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit

plugins_profile_root
The default Web server plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver/profiles/*profile_name* directory.

plugins_root
The default installation root directory for Web server plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V7/webserver directory.

plugins_user_data_root
The default Web server plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 7.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS7*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 7.0 product installed on the system is `QWAS7A`. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, `was.install.library`, and is located under the *app_server_root* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server Network Deployment is the /QIBM/UserData/WebSphere/AppServer/V7/ND/profiles/*profile_name* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS7`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

updi_root
The default installation root directory for the Update Installer for WebSphere Software is the /QIBM/ProdData/WebSphere/UpdateInstaller/V7/updi directory.

user_data_root
The default user data directory for WebSphere Application Server Network Deployment is the /QIBM/UserData/WebSphere/AppServer/V7/ND directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.