WebSphere® Application Server Network Deployment, Version 6

Using the administrative clients

Note

Before using this information, be sure to read the general information under “Notices” on page 581.

Compilation date: December 3, 2004

© Copyright International Business Machines Corporation 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2004

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 919-254-0206.

  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.

© Copyright IBM Corp. 2004

Chapter 1. Overview and new features for administering applications and their environments

This topic summarizes the contents and organization of the administration documentation, including links to conceptual overviews and descriptions of new features.

Sections in the administration documentation:

Setting up the application server environment
This section is for the administrator who is responsible for integrating application serving capabilities into an existing network environment. It looks at the product as part of a larger system, typically a production environment or realistic test environment. This section reiterates some installation and customization activities, including topology planning and creating product configurations. It carries the focus into the administrative realm, discussing port configuration and other network concerns. See also Overview and new features for installing an application server environment.

This information expands the topology planning discussion by describing how to set up and maintain logical administrative domains of cells and nodes, and how to balance workload through clustering and high availability configurations.

Chapter 3, "Using the administrative clients," on page 19
This section describes the many options available for administering your applications and the servers to which the applications are deployed. Options include the graphical administrative console; scripting with the wsadmin tool; programmatic administration using Java Management Extensions (JMX) and MBeans; and a wide array of command-line tools, including ANT.

Starting and stopping quick reference
This section summarizes what can be started and stopped, including applications and the application servers on which these applications are deployed.

Class loading
This section describes how to configure class loaders. It includes both configuration that is performed during application assembly (packaging) and configuration performed at the server. The product run-time environment uses class loaders to find and load new classes for an application. Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files.

Deploying and administering applications
This section describes how to deploy applications onto application servers, and then how to administer the deployed applications. It includes installing applications, starting applications, exporting application files, updating applications, removing applications, and other common tasks.

Administer WebSphere applications
This section provides administrative instructions that are specific to the various types of applications. For example, you can focus on administering your Web applications in their Web container; or aspects of Web services support; or the messaging or security subsystems.

Troubleshooting deployment
This section describes how to identify and handle a variety of problems encountered during development, assembly, and deployment activities.

Troubleshooting administration
This section describes how to identify and handle a variety of problems encountered during administrative activities.

© Copyright IBM Corp. 2004

Overview of administering applications and their environments

This topic provides links to conceptual overviews of administering your applications and application serving environment.

What is new for administrators

This topic provides an overview of new and changed features of system administration.

"Introduction: System administration" on page 3

This topic describes the administration of WebSphere Application Server, Version 6 products and the applications that run on them.

Presentations from Education on Demand

The following presentations provide a quick overview:

- System management architecture
- Administrative security
- Administrative clients overview
  - Start, stop, and monitor processes
  - Other commands
  - Browser-based administrative console
  - Scripting - wsadmin
  - Custom Java administrative client (JMX)
- Topologies and logical administrative domains
  - Resource scoping
  - Cells, deployment managers, and node agents
  - Build cells - Add and remove nodes
  - Manage node groups
- Applications and application resources
  - Application management overview
  - JDBC
  - Installing and uninstalling applications
  - Managed application resources - Enhanced EAR files
  - Fine grained application updates
- Servers
  - Server templates
  - Custom services
  - Manage Web server nodes
  - Application servers and cluster members
  - Creating cluster members
- Configuration management
  - Configuration repository
  - Configuration archives
  - File synchronization

Getting started with WebSphere Application Server

Note: If you prefer to browse PDF versions of this documentation using Adobe Reader, see the **Getting Started** PDF files that are available from www.ibm.com/software/webservers/appserv/infocenter.html.

IBM WebSphere Application Server products provide a next-generation application server on an industry-standard foundation. Each product addresses a distinct set of scenarios and needs. WebSphere Application Server, Version 6 product offerings are described in Packaging.

Planning

See Planning the installation (diagrams) for a description of typical scenarios for each WebSphere Application Server product.

Installing

See Task overview: Installing for a description of installing the WebSphere Application Server product and other installable components on the product disc.

Configuring

See Using the Profile creation wizard for a description of installing profiles that define a deployment manager, a managed node, or a stand-alone Application Server.

Migrating

See Migration and coexistence overview and Migrating and coexisting for a description of how to migrate applications and configuration data from a previous version of WebSphere Application Server.

Using the Samples Gallery

See Accessing the Samples (Samples Gallery) for a description of the set of Samples that ship with each product. The Samples demonstrate common Web application tasks.

Deploying applications

The information center describes a way to sample WebSphere Application Server functionality by quickly deploying Web components, such as servlets and JSP files. The method is not recommended as an official development method. See Fast paths for WebSphere Application Server to get started.

Introduction: System administration

Note: If you would prefer to browse PDF versions of this documentation using your Adobe Reader, see the **System Administration** PDF files available from www.ibm.com/software/webservers/appserv/infocenter.html.

A variety of tools are provided for administering the WebSphere Application Server product:

- **Console**

  The administrative console is a graphical interface that provides many features to guide you through deployment and systems administration tasks. Use it to explore available management options.

  For more information, refer to “Introduction: Administrative console” on page 4.

- **Administrative agents**

Servers, nodes and node agents, cells and the deployment manager are fundamental concepts in the administrative universe of the product. It is also important to understand the various processes in the administrative topology and the operating environment in which they apply.

For more information, refer to “Welcome to basic administrative architecture” on page 6.

- **Scripting**

  The WebSphere administrative (wsadmin) scripting program is a powerful, non-graphical command interpreter environment enabling you to run administrative operations in a scripting language. You can also submit scripting language programs to run. The wsadmin tool is intended for production environments and unattended operations.

  For more information, refer to “Introduction: Administrative scripting (wsadmin).”
- **Commands**

  Command-line tools are simple programs that you run from an operating system command-line prompt to perform specific tasks, as opposed to general purpose administration. Using the tools, you can start and stop application servers, check server status, add or remove nodes, and complete similar tasks.

  For more information, refer to “Introduction: Administrative commands” on page 5.
- **Programming**

  The product supports a Java programming interface for developing administrative programs. All of the administrative tools supplied with the product are written according to the API, which is based on the industry standard Java Management Extensions (JMX) specification.

  For more information, refer to “Introduction: Administrative programs” on page 5.
- **Data**

  Product configuration data resides in XML files that are manipulated by the previously-mentioned administrative tools.

  For more information, refer to “Introduction: Administrative configuration data” on page 6.

Introduction: Administrative console

The administrative console is a graphical interface for performing deployment and system administration tasks. It runs in your Web browser. Your actions in the console modify a set of XML configuration files.

You can use the console to perform tasks such as:

- Add, delete, start, and stop application servers
- Deploy new applications to a server
- Start and stop existing applications, and modify certain configurations
- Add and delete Java 2 Platform, Enterprise Edition (J2EE) resource providers for applications that require data access, mail, URLs, and so on
- Manage variables, shared libraries, and other configurations that can span multiple application servers
- Configure product security, including access to the administrative console
- Collect data for performance and troubleshooting purposes
- Find the product version information. It is located on the front page of the console.

“Starting and stopping the administrative console” on page 21 helps you begin using the console so that you can explore the available options. See also the **Reference > Administrator > Settings** section of the information center navigation. It lists the settings or properties you can configure.

Introduction: Administrative 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. 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.
- 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 global 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.
- Dynamic caching Includes caching topics, for example, creating, viewing and modifying a cache instance.
- Troubleshooting Provides information about how to troubleshoot using scripting. For example, tracing, thread dumps, profiles, and so on.
- Obtaining product information Includes tasks such as querying the product identification.
- 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.

Introduction: Administrative commands

Command-line tools are simple programs that you run from an operating system command-line prompt to perform specific tasks, as opposed to general purpose administration. Using the tools, you can start and stop application servers, check server status, add or remove nodes, and complete similar tasks.

See **Reference > Commands** in the information center navigation for the names and syntax of all the commands that are available with the product. A subset of these commands are particular to system administration purposes.

Introduction: Administrative programs

The product supports a Java programming interface for developing administrative programs. All of the administrative tools supplied with the product are written according to the API, which is based on the industry standard Java Management Extensions (JMX) specification. You can write a Java program that

performs any of the administrative features of the WebSphere Application Server administrative tools. You can also extend the basic WebSphere Application Server administrative system to include your own managed resources.

Introduction: Administrative configuration data

Administrative tasks typically involve defining new configurations of the product or performing operations on managed resources within the environment. IBM WebSphere Application Server configuration data is kept in files. Because all product configuration involves changing the content of those files, it is useful to know the structure and content of the configuration files.

The WebSphere Application Server product includes an implementation of the Java Management Extension (JMX) specification. All operations on managed resources in the product go through JMX functions. This setup means a more standard framework underlying your administrative operations as well as the ability to tap into the systems management infrastructure programmatically.

Welcome to basic administrative architecture

This article discusses basic concepts in the administrative architecture to help you understand system administration in a WebSphere Application Server environment. The fundamental concepts for WebSphere Application Server administration include software processes called servers, topological units referenced as nodes and cells, and the configuration repository used for storing configuration information.

Servers perform the actual running of the code. Several types of servers exist depending on the configuration. Each server runs in its own Java virtual machine (JVM). The application server is the primary run-time component in all WebSphere Application Server configurations. All WebSphere Application Server configurations can have one or more application servers. In some configurations, each application server functions as a separate entity. No workload distribution or common administration among application servers exists. In other configurations, workload can be distributed between servers and administration can be done from a central point.

A node is a logical group of WebSphere Application Server-managed server processes that share a common configuration repository. A node is associated with a single WebSphere Application Server profile. A WebSphere Application Server node does not necessarily have a one-to-one association with a system. One computer can host arbitrarily many nodes, but a node cannot span multiple computer systems. A node can contain zero or more application servers.

The configuration repository holds copies of the individual component configuration documents that define the configuration of a WebSphere Application Server environment. All configuration information is stored in .xml files.

A cell is a grouping of nodes into a single administrative domain. A cell can consist of multiple nodes, all administered from a deployment manager server. When a node becomes part of a cell (a federated node), a node agent server is installed on the node to work with the deployment manager server to manage the WebSphere Application Server environment on that node.

When a node is a standalone node, not part of a cell, the configuration repository is fully contained on the node. When a node is part of a cell, the configuration and application files for all nodes in the cell are centralized into a cell master configuration repository. This centralized repository is managed by the deployment manager server and synchronized to local copies that are held on each node. The local copy of the repository that is given to each node contains just the configuration information needed by that node, not the full configuration that is maintained by the deployment manager.

WebSphere Application Server types

This section discusses the three server types that interact to perform system administration.

Application Server: A WebSphere Application Server provides the functions that are required to support and host user applications. An application server runs on only one node, but one node can support many application servers.

Node agent: When a node is federated, a node agent is created and installed on that node. The node agent works with the deployment manager to perform administrative activities on the node.

Deployment manager: With the deployment manager, you can administer multiple application servers from one centralized manager. The deployment manager works with the node agent on each node to manage all the servers in a distributed topology.

The following diagram depicts the concepts that are discussed in this article.

The concepts that are discussed in this article form the basis of WebSphere Application Server administration. More detailed descriptions can be found in other sections.

Introduction: Servers

Application servers

Application servers provide the core functionality of the WebSphere Application Server product family. They extend the ability of a Web server to handle Web application requests, and much more. An application server enables a server to generate a dynamic, customized response to a client request.

For additional overview, refer to “Introduction: Application servers.”

Clusters

Workload management optimizes the distribution of client processing tasks. Incoming work requests are distributed to the application servers that can most effectively process the requests. Workload management also provides failover when servers are not available, improving application availability.

Clusters are sets of application servers that are managed together and participate in workload management. The servers that are members of a cluster can be on different host machines, as opposed to the servers that are part of the same node and must be located on the same host machine.

For additional overview, refer to “Introduction: Clusters” on page 9.

Introduction: Application servers

Overview

An application server is a Java Virtual Machine (JVM) that is running user applications. The application server collaborates with the Web server to return a dynamic, customized response to a client request. Application code, including servlets, JavaServer Pages (JSP) files, enterprise beans and their supporting classes, runs in an application server. Conforming to the Java 2 platform, Enterprise Edition (J2EE) component architecture, servlets and JSP files run in a Web container, and enterprise beans run in an Enterprise JavaBeans (EJB) container.

To begin creating and managing an application server, see Administering application servers.

You can define multiple application servers, each running its own JVM. Enhance the operation of an application server by using the following options:

- Configure transport chains to provide networking services to such functions as the service integration bus component of IBM service integration technologies, WebSphere Secure Caching Proxy, and the high availability manager core group bridge service. See Configuring transport chains for more information.
- Plug into an application server to define a hook point that runs when the server starts and shuts down. See Custom services for more information.
- Define command-line information that passes to a server when it starts or initializes. See “startServer command” on page 558 for more information.
- Tuning application sServers
- Enhance the performance of the application server JVM. See Using the JVM for more information.
- Use an Object Request Broker (ORB) for RMI/IIOP communication. See Managing object request brokers for more information.

Asynchronous messaging

The product supports asynchronous messaging based on the Java Messaging Service (JMS) of a JMS provider that conforms to the JMS specification version 1.1.

The JMS functions of the default messaging provider in WebSphere Application Server are served by one or more messaging engines (in a service integration bus) that runs within application servers.

In a deployment manager cell, there can be WebSphere Application Server version 5 nodes. If a version 5 node is configured to use V5 Default Messaging (the version 5 Embedded Messaging), there can be at most one JMS server on that node.

Generic Servers

In distributed platforms, the Generic Servers feature allows you create a generic server as an application server instance within the WebSphere Application Server administration, and associate it with a non-WebSphere server or process. The generic server can be associated with any server or process necessary to support the application server environment, including:
- A Java server
- A C or C++ server or process
- A CORBA server
- A Remote Method Invocation (RMI) server

After you define a generic server, you can use the Application Server administrative console to start, stop, and monitor the associated non-WebSphere server or process when stopping or starting the applications that rely on them.

For more information, refer to Creating generic servers.

Introduction: Web servers

In the WebSphere Application Server product, an application server works with a Web server to handle requests for dynamic content, such as servlets, from Web applications. SeeSupported Hardware and Software for this product for the most current information about supported Web servers.

The application server and Web server communicate using Web server plug-ins. Communicating with Web servers describes how to set up your Web server and Web server plug-in environment and how to create a Web server definition. The Web server definition associates a Web server with a previously defined managed or unmanaged node. After you define the Web server to a node, you can use the administrative console to perform the following functions for that Web server.

If the Web server is defined to a managed node, you can:
- Check the status of the Web server

- Generate a plug-in configuration file for that Web server.
- Propagate the plug-in configuration file after it is generated.

If the Web server is an IBM HTTP Server (IHS) and the IHS Administration server is installed and properly configured, you can also:
- Display the IBM HTTP Server Error log (error.log) and Access log (access.log) files.
- Start and stop the server.
- Display and edit the IBM HTTP Server configuration file (httpd.conf).

If the Web server it is defined to an unmanaged node, you can:
- Check the status of the Web server
- Generate a plug-in configuration file for that Web server.

If the Web server is an IBM HTTP Server (IHS) and the IHS Administration server is installed and properly configured, you can also:
- Display the IBM HTTP Server Error log (error.log) and Access log (access.log) files.
- Start and stop the server.
- Display and edit the IBM HTTP Server configuration file (httpd.conf).
- Propagate the plug-in configuration file after it is generated.

You can not propagate an updated plug-in configuration file to a non-IHS Web server that is defined to an unmanaged node. You must manually install an updated plug-in configuration file to a Web server that is defined to an unmanaged node. Web servers defined to an unmanaged node are typically remote Web servers. Remote Web servers are Web servers that are not located on the same machine as the WebSphere Application Server.

After you set up your Web server and Web server plug-in, whenever you deploy a Web application, you must specify a Web server as the deployment target that serves as a router for requests to the Web application. The configuration settings in the plug-in configuration file (plugin-cfg.xml) for each Web server are based on the applications that are routed through that Web server. If the Web server plug-in configuration service is enabled, a Web server plug-in's configuration file is automatically regenerated whenever a new application is associated with that Web server.

Note: Before starting the Web server, make sure you are authorized to run any Application Response Measurement (ARM) agent associated with that Web server.

Refer to your Web server documentation for information on how to administer that Web server. For tips on tuning your Web server plug-in, see Web server plug-in tuning tips.

Introduction: Clusters

Clusters are groups of servers that are managed together and participate in workload management. A cluster can contain nodes or individual application servers. A node is usually a physical computer system with a distinct host IP address that is running one or more application servers. Clusters can be grouped under the configuration of a cell, which logically associates many servers and clusters with different configurations and applications with one another depending on the discretion of the administrator and what makes sense in their organizational environments.

Clusters are responsible for balancing workload among servers. Servers that are a part of a cluster are called cluster *members*. When you install an application on a cluster, the application is automatically installed on each cluster member.

Because each cluster member contains the same applications, you can distribute client tasks in distributed platforms according to the capacities of the different machines by assigning weights to each server.

In distributed platforms, assigning weights to the servers in a cluster improves performance and failover. Tasks are assigned to servers that have the capacity to perform the task operations. If one server is unavailable to perform the task, it is assigned to another cluster member. This reassignment capability has obvious advantages over running a single application server that can become overloaded if too many requests are made.

Node groups bound clusters. All cluster members of a given cluster must be members of the same node group. For more information about clusters and node groups, see Clusters and node groups.

To learn more about clusters, see Clusters and workload management and Balancing workloads with clusters for more information.

Core groups

A group of clusters can be defined as a *core group*. All of the application servers defined as a member of one of the clusters included in a core group are automatically members of that core group. Individual application servers that are not members of a cluster can also be defined as a member of a core group. The use of core groups enables WebSphere Application Server to provide high availability for applications that must always be available to end users. You can also configure core groups to communicate with each other using the *core group bridge*. The core groups can communicate within the same cell or across cells.

To learn more about core groups, see Setting up a high availability environment.

Introduction: Environment

The environment of the product applies to the configuring of Web server plug-ins, variables, and objects that you want consistent throughout a cell.

Web servers

In the WebSphere Application Server product, an application server works with a Web server to handle requests for Web applications. The application Server and Web server communicate using a WebSphere HTTP plug-in for the Web server.

For more information, refer to “Introduction: Web servers” on page 8.

Cell-wide settings

Cell-wide settings are sets of configuration data that are stored in files in the cell directory. These configuration files are replicated to every node in the cell. Several different configuration settings apply to the entire cell. These settings include the definition of virtual hosts, shared libraries, and any variables that must be consistent throughout the entire cell.

For more information, refer to “Introduction: Cell-wide settings” on page 11.

Variables

A variable is a configuration property that can be used to provide a parameter for any value in the system. A variable has a name and a value to use in place of that name wherever the variable name is located within the system.

For more information, refer to Configuring WebSphere variables.

Introduction: Cell-wide settings

The configuration data for WebSphere Application Server is stored in XML files. The XML files exist in one of several directories in the configuration repository tree.

The directory in which a configuration file exists determines its scope, or how broadly or narrowly that data applies. Files in an individual server directory apply to that specific server only. Files in a node-level directory apply to every server on that node. Files in the cell directory apply to every server on every node within the entire cell.

Cell-wide settings are configuration files in the cell directory. The files are replicated to every node in the cell. Several different configuration settings apply to the entire cell. These settings include the definition of virtual hosts, shared libraries, and any variables that you want consistent throughout the entire cell.

Chapter 2. How do I administer applications and their environments?

- Establish the application serving environment
- Secure the application serving environment - see Security
- Set up Web access for applications
- Set up resources for applications to use
- Configure class loaders - see development and deployment
- Deploy and administer applications
- Use the administrative clients
- Troubleshoot deployment and administration

Establish the application serving environment

The following tasks involve establishing application serving capability in your network environment, whether you use single or clustered application servers. Servers can be grouped into administrative domains known as nodes and cells.

See also the overview:

- Version 6 topology and terminology

Create WebSphere profiles

Profiles are the files that define a stand-alone Application Server node, a managed node, or a deployment manager node. A profile also includes all of the files that the node can change.

Administer nodes

A node is a grouping of managed servers. Use this task to view information about and manage nodes.

Administer node agents

Node agents are administrative agents that represent a node to your system and manage the servers on that node. Node agents monitor application servers on a host system and route administrative requests to servers. A node agent is created automatically when a node is added to a cell.

Administer cells

When you installed the WebSphere Application Server Network Deployment product, a cell was created. A cell provides a way to group one or more nodes of your Network Deployment product. You probably will not need to reconfigure the cell. Use this task to view information about and manage a cell.

Administer configurations

© Copyright IBM Corp. 2004

Application server configuration files define the available application servers, their configurations, and their contents. You should periodically save changes to your administrative configuration. You can change the default locations of configuration files, as needed.

Configure remote file services

Configuration data for the WebSphere Application Server product resides in files. Two services help you reconfigure and otherwise manage these files: the file transfer service and file synchronization service. By default, the file transfer service is always configured and enabled at a node agent, so you do not need to take additional steps to configure this service. However, you might need to configure the file synchronization service.

Administer application servers

Create, configure, and operate application server processes. An application server configuration provides settings that control how an application server provides services for running enterprise applications and their components.

Administer other server types

One step in the process of creating an application server is to specify a template. A server template is used to define the configuration settings of the new server. You have the option of specifying the default server template or choosing a template that is based on a server that already exists. The default template will be used if you do not specify a different template when you create the server.

You can create other types of servers, to represent Web servers in your topology, or for other purposes. There are two types of *generic* servers: (1) Non-Java applications or processes, or (2) Java applications or processes. A *custom service* provides the ability to plug into a WebSphere application server to define a hook point that runs when the server starts and shuts down.

Balance workloads by clustering application servers

To monitor application servers and manage the workloads of servers, use server clusters and cluster members provided by the Network Deployment product.

Establishing high availability (HA) for failover

Planning ahead for high availability support is important in order to avoid the risk of a failure without failover coverage. The application server runtime of the infrastructure managed by a high availability manager includes such entities as cells and clusters. These components relate closely to core groups, high availability groups, and the policy that defines the high availability infrastructure. In a properly configured high availability environment, a high availability manager can reassess the environment it is managing and accept new components as they are added to the environment.

Administer the UDDI registry

The UDDI Registry is supplied as a J2EE application file, uddi.ear. Change its configuration properties using the assembly tools. You can use either the WebSphere Application Server administrative console or the Java Management Extensions (JMX) management interface to manage UDDI Registries.

Set up Web access for applications

These tasks involve enabling HTTP requests for applications on the application server.

- **Administer communication with Web servers (plug-ins)**

  The product provides plug-ins for supported Web servers, to enable the Web servers to pass requests to the application server, for applications running on the application server.

- **Administer HTTP sessions**

  Configure the service that the product provides for managing HTTP sessions: Session Manager.

- **Administer IBM HTTP Server Version 6.x**

  The product provides a complementary Web server with its own documentation that can be installed into the information center.

Set up resources for applications to use

Make a variety of resources available to your applications that are deployed on the application server.

- **Provide access to naming and directory resources (JNDI)**

  Configure naming. Naming is used by clients of WebSphere Application Server applications to obtain references to objects related to those applications, such as Enterprise JavaBeans (EJB) homes. These objects are bound into a mostly hierarchical structure, referred to as a name space. The name space structure consists of a set of name bindings, each consisting of a name relative to a specific context and the object bound with that name.

- **Provide access to relational databases (JDBC resources)**

  Configure data sources that applications use to access the data from databases.

- **Provide access to messaging resources (default messaging provider)**

  Use one of various ways to implement a messaging provider for use with WebSphere Application Server. A messaging provider enables use of the Java Messaging Service (JMS) and other message resources in the product.

- **Use IBM service integration technologies**

- **Establish workload balancing and high availability (HA) of messaging engines**

- **Access Service Integration (SI) bus resources**

Deploy and administer applications

These tasks involve deploying applications onto the application server, then administering the applications.

- **Install applications**

  Installable modules include enterprise archive (EAR), enterprise bean (EJB), Web archive (WAR), resource adapter (connector or RAR), and application client files.

- **Start and stop applications**

  You can start an application that is not running (has a status of Stopped) or stop an application that is running (has a status of Started).

- **Update applications**

  Update deployed applications or modules using the administrative console or **wsadmin** scripting. Learn which changes are candidates for hot deployment and dynamic reloading, in which you can make various changes to applications and their modules without having to stop the server and start it again.

- **Deploy applications rapidly (WebSphere Rapid Deployment)**

  Take advantage of new rapid deployment capabilities. WebSphere rapid deployment offers the following advantages: You do not need to assemble your J2EE application files prior to deployment. You do not need to use other installation tools mentioned in this table to deploy the files. Refer to the **Rapid deployment tools** documentation in the information center.

- **Enhanced EAR files**

- **Deploy and administer Web services applications**

  To deploy Web services that are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification, you need an enterprise application, also known as an enterprise archive (EAR) file that has been configured and enabled for Web services. You can use either the administrative console or the wsadmin scripting interface to deploy an EAR file.

Use the administrative clients

A variety of tools are provided for administering the product.

- **Choose an administrative client**

  Learn about and decide among the available administrative clients, including a graphical console, scripting (wsadmin), command line tools, and Java Management Extensions (JMX) programs.

- **Use the administrative console**

The administrative console is a Web-based tool that you use to administer the product. The administrative console supports a full range of product administrative activities.

Use scripting (wsadmin)

Scripting is a non-graphical alternative that you can use to configure and manage WebSphere Application Server. The WebSphere Application Server **wsadmin** tool provides the ability to run scripts. The tool supports a full range of product administrative activities.

See also:

- Start, stop, monitor processes
- Other administrative commands
- Custom Java administrative clients (JMX)

Troubleshoot deployment and administration

Troubleshoot problems that occur when you are deploying applications onto the application server, or when you are administering an established application serving environment.

Troubleshoot deployment

Troubleshoot problems that occur either during deployment or shortly afterwards, when you try to access an application that you just deployed for the first time.

Troubleshoot administration

Review some possible causes, based on the error you are seeing.

Chapter 3. Using the administrative clients

The product provides a variety of administrative clients for deploying and administering your applications and application serving environment, including configurations and logical administrative domains.

- Chapter 4, “Using the administrative console,” on page 21

  The administrative console is a graphical, browser-based tool.

- “Getting started with scripting” on page 37

  Scripting is a non-graphical alternative that you can use to configure and administer your applications and application serving environment. The WebSphere Application Server **wsadmin** tool provides the ability to run scripts. The wsadmin tool supports a full range of product administrative activities.

- Chapter 6, “Using Ant to automate tasks,” on page 513

  To support using Apache Ant with Java 2 Platform, Enterprise Edition (J2EE) applications running on IBM WebSphere Application Server, the product provides a copy of the Ant tool and a set of Ant tasks that extend the capabilities of Ant to include product-specific functions.

- Chapter 7, “Using administrative programs (JMX),” on page 515

  The product supports access to the administrative functions through a set of Java classes and methods, under the Java Management Extensions (JMX) specification. You can write a Java program that performs any of the administrative features of the other administrative clients. You also can extend the basic product administrative system to include your own managed resources.

- Chapter 8, “Using command line tools,” on page 557

  Several command-line tools are available that you can use to start, stop, and monitor WebSphere server processes and nodes. These tools work on local servers and nodes only. They cannot operate on a remote server or node.

© Copyright IBM Corp. 2004

Chapter 4. Using the administrative console

The administrative console is a Web-based tool that you use to manage the IBM WebSphere Application Server product as well as the Network Deployment product. The administrative console supports a full range of product administrative activities.

1. Distributed platforms: Start the server for the administrative console.For the Network Deployment product, the administrative console belongs to the deployment manager (dmgr) process, which you start with the **startmanager** command.
2. Access the administrative console.
3. Change the session timeout for the administrative console. (Optional)
4. Browse the administrative console.
5. Specify console preferences.
6. Access help.

Starting and stopping the administrative console

This topic describes how to set up the administrative console environment, to access the administrative console, and to log out of the administrative console.

To access the administrative console, you must start it and then log in. After you finish working in the console, save your work and log out.

1. Start the administrative console.
   a. Distributed platforms: Verify that the administrative console runs on the `server1` application server for the WebSphere base product. Verify that the administrative console runs on the deployment manager for the Network Deployment product. Use the wasadmin **startManager** command to start the deployment manager.
   b. Enable cookies in the Web browser that you use to access the administrative console for the administrative console to work correctly.
   c. Distributed platforms: In the same Web browser, type `http://`*your_fully_qualified_server_name*`:9060/ibm/console`, where *your_fully_qualified_server_name* is the fully qualified host name for the machine that contains the administrative server. When the administrative console is on the local machine, *your_fully_qualified_server_name* can be `localhost` unless security is enabled. On Windows platforms, use the actual host name if `localhost` is not recognized. If security is enabled, your request is redirected to `https://`*your_fully_qualified_server_name*`:9043/ibm/console`, where *your_fully_qualified_server_name* is the fully qualified host name for the machine that contains the administrative server.

      For a listing of supported Web browsers, see WebSphere Application Server system requirements at

      `http://www.ibm.com/software/webservers/appserv/doc/latest/prereq.html`

      The Web address appears on two lines for printing purposes. Enter the Web address on one line in your browser.
   d. Wait for the console to load into the browser. A Login page is displayed after the console starts.

   If you cannot start the administrative console because the console port conflicts with an application that is already running on the machine, change the port number in the *install_root*`/profiles/`*profile name*`/config/cells/`*cell_name*`/ nodes/`*node_name*`/servers/`*server_name*`/server.xml` file and the *install_root*`/profiles/`*profile name*`/config/cells/`*cell_name*`/virtualhosts.xml` files. Change all the occurrences of port 9060 (or the port that is selected during profile creation for WebSphere Application Server) to the port for the console. Alternatively, shut down the other application that uses the conflicting port before starting the WebSphere Application Server product.

© Copyright IBM Corp. 2004

2. Log into the console.
   a. Enter your user name or user ID.

      The user ID lasts only for the duration of the session for which it was used to log in.

      Changes made to server configurations are saved to the user ID. Server configurations also are saved to the user ID if a session timeout occurs.

      If you enter an ID that is already in use (and in session), you are prompted to do one of the following actions:
      - Force the existing user ID out of session. The configuration file that is used by the existing user ID is saved in the temporary area.
      - Wait for the existing user ID to log out or time out of the session.
      - Specify a different user ID.

   b. If the console is secure, you must also enter a password for the user name. The console is secure if someone has taken the following actions for the console:
      - Specified security user IDs and passwords
      - Enabled global security

   c. Click **OK**.
3. Stop the administrative console. Click **System administration > Save changes to Master Repository > Save** to save work. Then click **Logout** to exit the console.

   If you close the browser before saving your work, when you next log in under the same user ID, you can recover any unsaved changes.

Login settings

Use this page to specify the user for the WebSphere Application Server administrative console. If you are using global security, then you must also specify a password.

When you specify a user, you can resume work done previously with the product. After you type in a user ID, and password if you are using global security, click **OK** to proceed to the next page and access the administrative console.

To view this page, start the administrative console.

Logging into the administrative console

When you log into the administrative console, you can optionally specify a user ID if the console is not secure. If the administrative console is secure, you must specify a user ID and password.

User ID

Specifies a string that identifies the user. The user ID must be unique to the administrative server. Concurrent administrative console sessions must use unique user IDs.

Work that you do with the product and then save before exiting the product is saved to a configuration that is identified by the user ID that you enter. To later access work done under that user ID, specify the same user ID in the Login page.

Data type String

Password

If you use global security, specify a password.

Resolving conflicts during login

Conflicts can result if you log into the administrative console with a user ID that is already in use.

Another user is currently logged in with the same user name

Specifies whether to log out the user and to continue work with the user ID that is specified, or to return to the Login page and specify a different user ID, or wait for the user to log out.

This field is displayed if:

- The user closed a Web browser while browsing the administrative console and did not first log out, then opened a new browser and tried to access the administrative console with the same user ID.
- The user opened a Web browser to access the administrative console while accessing the administrative console in another open Web browser with the same user ID.
- The user opens a Web browser and attempts to log into the console with the same user ID that is already in use by another user who logged into the console from another Web browser on another computer.

Recovering prior changes

You can either recover changes that you made to the configuration from a prior session or use the master configuration. The default is to recover changes from a prior session.

Recover changes made in a prior session

When enabled, this setting specifies that you want to use the same administrative configuration used for the last user's session. This option recovers changes made by the user since the last saving of the administrative configuration for the user's session.

This field is displayed only if the user changed the administrative configuration and then logged out without saving the changes.

Work with the master configuration

When enabled, this setting specifies to use the default administrative configuration instead of the configuration that was last used for the user's session. Changes that are made to the user's session since the last saving of the administrative configuration are lost.

This field is displayed only if the user changed the administrative configuration and then logged out without saving the changes.

Resolving login failures

When the administrative console is enabled with global security, you must type in a valid user ID and password. If the user ID, password, or both are not valid, you receive the following message:

```
Unable to process login. Please check User ID and password and try again.
```

Resolve the problem by entering a valid user ID and password as defined in the WebSphere Application Server security documentation.

Save changes to the master configuration

Use this page to update the master repository with your administrative console changes, to discard your administrative console changes and continue working with the master repository, or to continue working with your administrative console changes that are not saved to the master repository.

Until you save changes to the master repository, the administrative console uses a local workspace to track your changes.

Total changed documents

Specifies the total number of documents that you changed for your session, but that are not saved to the master repository. By clicking the +/- toggle key, you can see additional information about the changed documents:

- **Changed items**

  When you change your local configuration, each path and configuration file that you can apply the update to in the master repository is displayed in the list.
- **Status**

  Can contain the following options:
  - **Added:** If you save your changes to the master repository, a new configuration file is created on the indicated path.
  - **Updated:** If you save your changes to the master repository, an existing configuration file is updated on the indicated path.
  - **Deleted:** If you save your changes to the master repository, an existing configuration file is deleted on the indicated path.

Synchronize changes with nodes

Specifies whether you want to force node synchronization at the time that you save your changes to the master repository rather than when node synchronization normally occurs.

Setting the session timeout for the administrative console

This topic describes how to change the session timeout from the default value for the administrative console.

Ensure that you have the proper permissions to change the `${WAS_HOME}/systemApps/adminconsole.ear/deployment.xml` file.

Determine whether the default session timeout value of 30 minutes is acceptable. Some reasons that you might change the default value are:

- Users in secure environments might need shorter session timeout periods to ensure security, encase they leave their machine and forget to log off the console.
- Users might need longer session timeout periods if they respond slower than typical users for accessibility reasons.
- Users in secure environments might not want the administrative console timeout value to conflict with Lightweight Third-Party Authentication (LTPA) cookie timeouts

Do the following actions to change the timeout value:

1. Edit the `${WAS_HOME}/systemApps/adminconsole.ear/deployment.xml` file in a text editor.
2. Locate the xml statement `<tuningParams xmi:id="TuningParams_1088453565469" maxInMemorySessionCount="1000" allowOverflow="true" writeFrequency="TIME_BASED_WRITE" writeInterval="10" writeContents="ONLY_UPDATED_ATTRIBUTES" invalidationTimeout="30">`
3. Change the invalidationTimeout value to the desired session timeout. The default is 30.
4. Save the `${WAS_HOME}/systemApps/adminconsole.ear/deployment.xml` file.
5. Restart the console.

Once you restart the console, the change takes effect.

Manage WebSphere Application Server through the administrative console.

Administrative console areas

Use the administrative console to create and manage objects in the WebSphere Application Server configuration such as resources, applications, and servers. Additionally, use the administrative console to view product messages. This topic describes the main areas that display on the administrative console.

To view the administrative console, ensure that the application server for the administrative console is running. Point a Web browser at the Web address for the administrative console, enter your user ID and, if needed, a password on the Login page.

You can resize the width of the navigation tree and workspace simultaneously by dragging the border between them to the left or the right. The change in width does not persist between administrative console user sessions.

The console has the following main areas.

Taskbar

The taskbar offers options for logging out of the console, accessing product information, and accessing support.

Navigation tree

The navigation tree on the left side of the console offers links to console pages that you use to create and manage components in a WebSphere Application Server administrative cell.

Click a plus sign (**+**) beside a tree folder or item to expand the tree for the folder or item. Click a minus sign (**-**) to collapse the tree for the folder or item. Click an item in the tree view to toggle its state between expanded and collapsed.

Workspace

The workspace on the right side of the console contains pages that you use to create and manage configuration objects such as servers and resources. Click links in the navigation tree to view the different types of configured objects. Within the workspace, click configured objects to view their configurations, run-time status, and options. Click **Welcome** in the navigation tree to display the workspace Home page, which contains links to information on using the WebSphere Application Server product.

Administrative console buttons

This page describes the button choices that are available on various pages of the administrative console, depending on which product features you enable.

- **Check all.** Selects each resource that is listed on the administrative console panel, in preparation for performing an action against the selected resources.
- **Uncheck all.** Removes all the listed resources from each selection so that no action is performed against any of the resources.
- **Filter the view.** Produces a dialog box for specifying the resources to view in the table on this administrative console page.

  Hide the filter view. Hides the dialog box for specifying the resources to view in the table on this administrative console page.

  When you produce the dialog box, select the column to filter and enter the filter criteria.

  Column to filter

  Select the column to filter from the drop-down list. When you apply the filter, only those items in the selected column that meet the filter criteria are displayed.

  For example, select **Names** to enter criteria by which to filter application server names.

  Filter criteria

  Enter a string that must be found in the name of a collection entry to qualify the entry to display

in the collection table. The string can contain percent sign (%), asterisk (*), or question mark (?) symbols as wildcard characters. For example, enter `*App*` to find any application server whose name contains the string `App`.

Prefix each of the following characters ( ) ^ * % { } \ + $ with a backslash (\) so that the regular expression engine performing the search correctly matches the search criteria. For example, to search for all Java DataBase Connectivity (JDBC) providers containing (XA) in the provider name, specify the following string:

```
*\(XA\)
```

- **Clear filter criteria.** Clears your filter changes and restores the most recently saved values.
- **Abort.** Stops a transaction that is not yet in the prepared state. All operations that the transaction completed are undone.
- **Activate.** Activates a group member.
- **Add.** Adds the selected or typed item to a list, or produces a dialog for adding an item to a list.
- **Add Node.** Displays the Add Node page, in which you specify the host name and SOAP connector port for a node that you want added to a cell.
- **Apply.** Saves your changes to a page without exiting the page.
- **Back.** Displays the previous page or item in a sequence. The administrative console does not support using the Back and Forward options of a browser, which can cause intermittent problems. Use Back or Cancel on the administrative console panels instead.
- **Balance.** Balances active members in high availability groups across servers that host the high availability groups. The administrator must first determine which groups have active members and select those groups before selecting Balance.
- **Browse.** Opens a dialog that enables you to look for a file on your system.
- **Calculate groups.** Calculates the number of high availability groups that are returned based on the match set.
- **Cancel.** Exits the current page or dialog, discarding unsaved changes. The administrative console does not support using the Back and Forward options of a browser, which can cause intermittent problems. Use Cancel on the administrative console panels instead.
- **Change.** In the context of security, you can search the user registry for a user ID for an application to run under. In the context of container properties, you can change the data source that the container is using.
- **Clear.** Clears your changes and restores the most recently saved values.
- **Clear selections.** Clears any selected cells in the tables on this tabbed page.
- **Close.** Exits the dialog.
- **Commit.** Releases all locks that are held by a prepared transaction and forces the transaction to commit.
- **Copy.** Creates copies of the selected application servers.
- **Create.** Saves your changes to all the tabbed pages in a dialog and exits the dialog.
- **Create tables.** Develops scheduler database tables.
- **Deactivate.** Deactivates a group member. The group member must be in the active state to be deactivated. The deactivate option causes the group member to move to the idle state. The group policy overrides which members are activated and deactivated for a group. The policy is enforced for every member state change. If the deactivate option conflicts with the group policy, the policy resets who is the active member of the group.
- **Delete.** Removes the selected instance.
- **Details.** Shows the details about a transaction.
- **Disable.** Disables a group or group member. When you disable a group or group member, the active group or group member is first deactivated. If the deactivate option is successful, the group or group member moves to the disable state. A disabled group or group member cannot be activated.
- **Done.** Saves your changes to all the tabbed pages in a dialog and exits the dialog.
- **Down.** Moves through a list.
- **Drop tables.** Removes scheduler database tables.
- **Dump.** Activates a dump of a traced application server.
- **Edit.** Lets you edit the selected item in a list, or produces a dialog box for editing the item.
- **Enable.** Enables a group or a group member.

- **Export.** Accesses a page for exporting enterprise archive (EAR) files for an enterprise application.
- **Export DDL.** Accesses a page for exporting data definition language (DDL) files for an enterprise application.
- **Export Keys.** Exports Lightweight Third-Party Authentication (LTPA) keys to other domains.
- **Export route table.** Exports the route table information for a selected cluster to a binary file in the configuration.
- **Filter.** Produces a dialog box for specifying the resources to view in the tables on this tabbed page.
- **Finish.** Forces a transaction to finish, regardless of whether its outcome has been reported to all participating applications.
- **First.** Displays the first record in a series of records.
- **Full resynchronize.** Synchronizes the user's configuration immediately. Click full resynchronize on the Nodes page if automatic configuration synchronization is disabled, or if the synchronization interval is set to a long time, and a configuration change is made to the cell repository that needs to be replicated to that node. Clicking this option clears all synchronization optimization settings and performs configuration synchronization again, so no mismatches occur between node and cell configuration after this operation is performed. This operation can take awhile to perform.
- **Force delete.** Forces the removal of a node that is not removed properly from the cell in the master repository. The **Remove node** action is preferred over the **Force delete** action to delete a node from the configuration. If you click **Force delete**, but the node still exists in the configuration, uninstall the node or run the **removeNode** command by using the -force parameter on that node. Force delete action is equivalent to running the cleanupNode command at the deployment manager.
- **Generate keys.** Generates new LTPA keys. When security is turned on for the first time with LTPA as the authentication mechanism, LTPA keys are automatically generated with the password entered in the panel. To generated new keys, use this option after the server is up with security turned on. Clicking this option generates the keys and propagates them to all active servers (cell, node, and application servers). The new keys can be used to encrypt and decrypt the LTPA tokens. Click **Save** on the console taskbar to save the new keys and the password in the repository.
- **Immediate stop.** Stops the server, but bypasses the normal server quiesce process that supports in-flight requests to complete before shutting down the entire server process. This shutdown mode is faster than the normal server stop processing, but some application clients can receive exceptions.
- **Import keys.** Imports new LTPA keys from other domains. To support single signon (SSO) in WebSphere Application Server across multiple WebSphere domains (cells), share LTPA keys and a password among the domains. After exporting the keys from one of the cells into a file, click this option to import the keys into all the active servers (cell, node, and application servers). The new keys can be used to encrypt and decrypt the LTPA token. Click **Save** on the console taskbar to save the new keys and the password in the repository.
- **Install.** Displays the Preparing for application installation page, which you use to deploy an application, an enterprise bean, or a Web component onto an application server.
- **Install RAR.** Opens a dialog that is used to install a Java 2 Platform, Enterprise Edition Connector Architecture (JCA) connector and to create a resource adapter.
- **Manage transactions.** Displays a list of active transactions running on a server. You can forcibly finish any transaction that has stopped processing because a transactional resource is not available.
- **Modify.** Opens a dialog that is used to change a specification.
- **Move.** Moves the selected application servers to a different location in the administrative cell. When prompted, specify the target location.
- **Move down.** Moves downward through a list.
- **Move up.** Moves upward through a list.
- **New.** Displays a page that you use to define a new instance. For example, clicking **New** on the Application Servers page displays a page on which you can configure a new application server.
- **Next.** Displays the next page, frame, or item in a sequence.
- **OK.** Saves your changes and exits the page.
- **Ping.** Attempts to contact selected application servers.
- **Previous.** Displays the previous page, frame, or item in a sequence.
- **Quit.** Exits a dialog box and discards any unsaved changes.
- **Refresh.** Refreshes the view of data for instances that are currently listed on this tabbed page.

- **Regenerate encryption key.** Regenerates a key for global data replication. If you are using the DES or TRIPLE_DES encryption type, regenerate a key at regular intervals (for example, monthly) to enhance security.
- **Remove.** Deletes the selected item.
- **Remove file.** Removes the specified file from the selected application or module.
- **Remove node.** Deletes the selected node.
- **Reset.** Clears your changes on the tab or page and restores the most recently saved values.
- **Restart.** Stops the selected objects and starts them again.
- **Restart all servers on node.** Stops all application servers on the node and starts them again.
- **Retrieve new.** Retrieves a new record.
- **Rollout update.** Sequentially updates an application that is installed on multiple cluster members across a cluster. After you update application files or a configuration, click **Rollout update** to install the configuration or the updated files for an application on all the cluster members of a cluster on which the application is installed. The Rollout update option applies the following steps to each cluster member in sequence:
  1. Saves an updated configuration.
  2. Stops the cluster member.
  3. Updates the application on the node by synchronizing the configuration.
  4. Restarts the cluster member.

  This action enables you to update an application on multiple cluster members while providing continuous availability of the application.
- **Save.** Saves the changes in your local configuration to the master configuration.
- **Select.** For resource analysis, lets you select a scope in which to monitor resources.
- **Set.** Saves your changes to settings in a dialog.
- **Settings.** Displays a dialog for editing servlet-related resource settings.
- **Settings in use.** Displays a dialog showing the settings in use.
- **Show groups.** Displays a collection of high availability groups, based on the match set.
- **Show servers.** Displays a collection of servers that are contained in the high availability groups that match the match set.
- **Start.** In the context of application servers, starts selected application servers. In the context of data collection, starts collecting data for the tables on this tabbed page.
- **Stop.** In the context of server components such as application servers, stops the selected server components. In the context of a data collection, stops collecting data for the tables on a tabbed page. In the context of nodes, stops servers on the selected nodes. In the context of deployment managers, stops the deployment manager server.
- **Synchronize.** Synchronizes the user's configuration immediately. Click Synchronize on the Nodes page if automatic configuration synchronization is disabled, or if the synchronization interval is set to a long time, and a configuration change is made to the cell repository that needs replicating to that node. A node synchronization operation is performed using the normal synchronization optimization algorithm. This operation is fast, but might not fix problems from manual file edits that occur on the node. It is possible for the node and cell configuration to be out of synchronization after this operation is performed. If problems persist, use Full Resynchronize.
- **Terminate.** Deletes the Application Server process or another process that cannot be stopped by the **Stop** or **Immediate Stop** commands. Some application clients can receive exceptions. Always attempt an immediate stop before using this option.
- **Test connection** After you define and save a data source, you can select this option to ensure that the parameters in the data source definition are correct. On the Collection panel, you can select multiple data sources and test them simultaneously.
- **Uninstall.** Deletes a deployed application from the WebSphere Application Server configuration repository. Also deletes application binary files from the file system.
- **Update.** Replaces an application that is deployed on a server with an updated application. As part of the updating, you might need to complete steps on the Preparing for application installation and Update application pages.
- **Update resource list.** Updates the data on a table. Discovers and adds new instances to the table.

- **Use cell CSI.** Enables Object Management Group (OMG) Common Secure Interoperability (CSI) protocol.
- **Use cell SAS.** Enables IBM Secure Authentication Service (SAS).
- **Use cell Security.** Enables cell security.
- **Verify tables.** Validates the mapping between the table names, scheduler resource, and data sources.
- **View.** Opens a dialog on a file.

Administrative console page features

This topic provides information about the basic elements of an administrative console page, such as the various tabs.

Administrative console pages are arranged in a few basic patterns. Understanding their layout and behavior will help you use them more easily.

Collection pages

Use collection pages to manage a collection of existing administrative objects. A collection page typically contains one or more of the following elements:

Scope and Preferences
: These are described in “Administrative console scope settings” on page 33 and “Administrative console preference settings” on page 32.

Table of existing objects
: The table displays existing administrative objects of the type specified by the collection page. The table columns summarize the values of the key settings for these objects. If no objects exist yet, an empty table is displayed. Use the available buttons to create a new object.

Buttons for performing actions
: The available buttons are described on the Administrative console buttons help panel. In most cases, you need to select one or more of the objects in the table, then click a button. The action will be applied to the selected objects.

Sort toggle buttons
: Following column headings in the table are icons for sort ascending (^) and sort descending (v). By default, items such as names are sorted in descending order (alphabetically). To enable another sorting order, click on the icons for the column whose items you want sorted.

Detail pages

Use detail pages to configure specific administrative objects, such as an application server. A detail page typically contains one or more of the following elements:

Configuration tabbed page
: This tabbed page is for modifying the configuration of an administrative object. Each configuration page has a set of general properties specific to the administrative object. Other sets of properties display on the page, but vary depending on the administrative object.

Runtime tabbed page
: This tabbed page displays the configuration that is currently in use for the administrative object. It is read-only in most cases. Some detail pages do not have runtime tabs.

Local Topology tabbed page
: This tabbed page displays the topology that is currently in use for the administrative object. View the topology by expanding and collapsing the different levels of the topology. Some detail pages do not have local topology tabs.

Buttons for performing actions
: Buttons to perform specific actions display on the configuration tabbed page and the runtime tabbed page. The displayed buttons vary based on the administrative object. The available buttons are described on the Administrative console buttons help panel.

Wizard pages

Use wizard pages to complete a configuration process comprised of several steps. Be aware that wizards show or hide certain steps depending on the characteristics of the specific object you are configuring.

Administrative console navigation tree actions

Use the navigation tree of the administrative console to access pages for creating and managing servers, applications, resources, and other components.

To view the navigation tree, go to the WebSphere Application Server administrative console and look at the tree on the left side of the console. The tree provides navigation to configuration tasks and run-time information. The main topics available on the navigation tree are detailed in the following section. To use the tree, expand a main topic and select an item from the expanded list to display a page on which you can perform the administrative task.

Servers

Configure application servers, clusters, generic servers, Web servers, and core groups.

Applications

Install applications onto servers and manage the installed applications.

Resources

Configure resources and to view information on resources that exist in the administrative cell.

Security

Access the Security Center, which you use to secure applications and servers.

Environment

Configure hosts, WebSphere Application Server variables, and other components.

System Administration

Configure console settings, and manage components and users of a Network Deployment product.

Troubleshooting

Check for configuration errors and problems, view log files, and enable and disable tracing on a distributed platform.

Monitoring and Tuning

Monitor and tune your Application Server performance and analyze performance data.

Service Integration

Iimplement message-oriented and service-oriented applications.

UDDI

Publish and discover information about Web services.

Administrative console taskbar actions

Use the taskbar of the administrative console to log out of the administrative console and to access the console help.

To view the taskbar, go to the WebSphere Application Server administrative console and look at the horizontal bar near the top of the console. The taskbar provides the following actions.

Logout

Logs you out of the administrative console session and displays the Login page. If you made changes to the administrative configuration since last saving the configuration to the master repository, the Save page is displayed before returning to the Login page.

- Click **Save** to save the changes to the master repository.

- Click **Discard** to exit the session without saving changes.
- Click **Logout** to exit the session without saving changes but with the opportunity to recover your changes when you return to the console.

Help
Opens a new Web browser to online help for the WebSphere Application Server product.

Support
Displays support links that vary based on the products that extend the WebSphere Application Server. Use the support page to access product information such as Frequently Asked Questions (FAQs), technical notes (Technotes), hints and tips, and news. You can additionally install the Support Advisor Search application so that when you click on the support link, a new Web browser that contains the Support Advisor Search application opens. The Support Advisor Search application displays the support links on the support page, but additionally provides federated search capabilities into IBM knowledge databases.

Specifying console preferences

Use this topic to customize how much data displays on an administrative console panel.

Throughout the administrative console are pages that have Preferences fields, Scope fields, and Filter radio buttons. By selecting these fields and radio buttons you can customize how much data is shown.

For example, examine the Preferences field for the Enterprise Applications page:

1. Go to the navigation tree of the administrative console and click **Applications > Enterprise Applications**.
2. Expand **Preferences**.
3. For the **Maximum rows** field, specify the maximum number of rows to display when the collection is large. The default is 20. Rows that exceed the maximum number display on subsequent pages.
4. Select **Retain filter criteria** if you want to retain the last filter criteria that is entered in the filter function. When you return to the Applications page, the page initially uses the retained filter criteria to display the collection of applications in the table following the preferences. Otherwise, clear **Retain filter criteria** and the last filter criteria is not retained.
5. Click **Apply** to apply your selections or click **Reset** to return to the default values. The default is not to enable (not have a check mark beside) **Retain filter criteria**.

Other pages have similar fields and radio buttons that you can use to specify console preferences. While Preferences fields, Scope fields, and Filter buttons control how much data is shown in the console, the **Preferences** option controls general behavior of the console. Click **System administration > Console settings > Preferences** to view the Preferences page.

Preferences settings

Use the Preferences page to specify whether you want the administrative console workspace to refresh automatically after changes, the default scope to be the administrative console node, confirmation dialogs to display, and the workspace banner and descriptions to display.

To view this administrative console page, click **System administration > Console settings > Preferences**.

Turn on workSpace auto-refresh
Specifies whether you want the administrative console workspace to redraw automatically after the administrative configuration changes.

The default is for the workspace to redraw automatically. If you direct the console to create a new instance of, for example, an application server, the Application Servers page refreshes automatically and shows the new server name in the collection of servers.

Specifying that the workspace not redraw automatically means that you must access a page again by clicking the console navigation tree or links on collection pages to see the changes that are made to the administrative configuration.

Default true (selected)

No confirmation on workspace discard
Specifies whether the confirmation dialog is displayed after a request is receive to discard the workspace. The default is to display confirmation dialogs.

Default false (cleared)

Use default scope (administrative console node)
Specifies whether the default scope is the administrative console node. The default scope not is not the console node.

Default false (cleared)

Show banner
Specifies whether the WebSphere Application Server banner along the top of the administrative console is displayed. The default is for the banner to display.

Default true (selected)

Show Descriptions
Specifies whether information on the right of the console is shown. The default is to show the information.

Data type Boolean
Default true

Administrative console preference settings
Use the preference settings to specify how you want information displayed on an administrative console page.

Maximum rows
Indicates the maximum number of rows to display per page when the collection is large.

Filter history
Indicates whether to use the same filter criteria to display this page the next time you visit it.

Select the **Retain filter criteria** check box to retain the last filter criteria entered. When you return to the page, retained filter criteria control the application collection that is displayed n the table.

Show confirmation for stop command
Select the check box if you want a confirmation that the **stop** command is successful.

Show confirmation for immediate stop command
Select the check box if you want a confirmation that the **immediate stop** command is successful.

Show confirmation for terminate command
Select the check box if you want a confirmation that the **terminate** command is successful.

Administrative console scope settings
Use this page to specify the level at which a resource is visible on the administrative console panel. A resource can be visible in the administrative console collection table at the cell, node, cluster, or server scope. By changing the value for Scope you can see other variables that apply to a resource and might change the contents of the collection table.

Click **Browse** next to a field to see choices for limiting the scope of the field. If a field is read-only, you cannot change the scope. For example, if only one server exists, you cannot switch the scope to a different server.

You always create resources at the current scope that is selected in the administrative console panel, even though the resources might be visible at more than one scope.

Resources such as JDBC providers, namespace bindings, or shared libraries can be defined at multiple scopes. Resources that are defined at more specific scopes override duplicate resources that are defined at more general scopes.

- The application scope has precedence over all the scopes.
- The server scope has precedence over the node, cell, and cluster scopes.
- The cluster scope has precedence over the node and cell scopes.
- The node scope has precedence over the cell scope.

Despite the scope of a defined resource, the resource properties only apply at an individual server level. For example, if you define the scope of a data source at the cell level, all the users in that cell can look up and use that data source, which is unique within that cell. However, resource property settings are local to each server in the cell. For example, if you define the maximum connections as 10, then each server in that cell can have 10 connections.

The cell scope is the most general scope and does not override any other scope. The recommendation is that you generally specify a more specific scope than the cell scope. When you define a resource at a more specific scope, you provide greater isolation for the resource. When you define a resource at a more general scope, you provide less isolation. Greater exposure to cross-application conflicts occur for a resource that you define at a more general scope.

Cell Limits the visibility to all servers on the named cell. The resource factories within the cell scope are:
- Defined for all servers within this cell
- Overridden by any resource factories that are defined within application, server, cluster and node scopes that are in this cell and have the same Java Naming and Directory Interface (JNDI) name

The resource providers that are required by the resource factories must be installed on every node within the cell before applications can bind or use them.

Cluster

Limits the visibility to all the servers on the named cluster. All cluster members must at least be at Version 6 to use cluster scope for the cluster. The resource factories that are defined within the cluster scope:
- Are available for all the members of this cluster to use
- Override any resource factories that have the same JNDI name that is defined within the cell scope

The resource factories that are defined within the cell scope are available for this cluster to use, in addition to the resource factories, that are defined within this cluster scope.

Node Limits the visibility to all the servers on the named node. The node scope is the default scope for most resource types. The resource factories that are defined within the node scope:
- Are available for servers on this node to use
- Override any resource factories that have the same JNDI name defined within the cell scope

The resource factories that are defined within the cell scope are available for servers on this node to use, in addition to the resource factories that are defined within this node scope.

Server
Limits the visibility to the named server. The server scope is the most specific scope for defining resources. The resource factories that are defined within the server scope:
- Are available for applications that are deployed on this server
- Override any resource factories that have the same JNDI name defined within the node and cell scopes

The resource factories that are defined within the node and cell scopes are available for this server to use, in addition to the resource factories that are defined within this server scope.

Application
Limits the visibility to the named application. Application scope resources cannot be configured from the console. Use the WebSphere Application Server Toolkit (AST) or the wsadmin tool to view or modify the application scope resource configuration. The resource factories that are defined within the application scope are available for this application to use only. The application scope overrides all other scopes.

You can configure resources and WebSphere Application Server variables under all five scopes. You can configure namespace bindings and shared libraries only under cell, node, and server scopes.

Accessing help and product information from the administrative console

This topic describes how to use administrative console help and how to link to product documentation from the administrative console.

You must have a connection to the Internet to access information about WebSphere Application Server from the Welcome page of the administrative console.

All of the helps panels that you can access from the administrative console, you can access from the WebSphere Application Server Information Center. This article describes how to access the help panels, the information center, and other product documentation from the administrative console.

- Click **Welcome** on the administrative console navigation tree. In the workspace to the right of the navigation tree, select the appropriate links to access the WebSphere Application Server Information Center, the WebSphere Application Server product information, and the WebSphere Application Server technical information on developerWorks.
- Access help in the following ways:
  - Click **Help** on the administrative console task bar to open a new Web browser for online help.
    - Click on the **Help index** tab and select from the list of help panels to view administrative console help information.
    - Click on the **Search** tab, provide search terms, and then click **Search**. Under Results, select a help panel that contains the search information.
  - Click the **?** icon on the task bar for the particular administrative console panel to open a new Web browser and view the help panel for the corresponding administrative console panel. The help panel is displayed in the Help index for the administrative console.
  - In the help portal that is on the right side of the administrative console panel, do one or all of the following tasks:

- Click a field label or a list marker in the administrative console panel for the help to display under Field help. Alternatively, place the cursor over the field label or the list marker for the corresponding help to display at the cursor.
- Click the link under Page help to access the help panel for the administrative console panel. The help panel is the same help panel that displays when you click the **?** icon.
- Expand the task help to view related tasks.

You can continue to access help information from the administrative console. Alternatively, you can access the help information from the WebSphere Application Server Information Center.

You can continue to access the WebSphere Application Server Information Center, the WebSphere Application Server product information, and the WebSphere Application Server technical information on developerWorks from the administrative console. Alternatively you can access the information from the IBM Web site.

Administrative console: Resources for learning

Use the following links to find relevant supplemental information about the IBM WebSphere Application Server administrative console. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information:

Administration

- IBM WebSphere Application Server Redbooks

  This site contains a listing of all WebSphere Application Server Redbooks.
- IBM developerWorks WebSphere

  This site is the home of technical information for developers working with WebSphere products. You can download WebSphere software, take a fast path to developerWorks zones, such as VisualAge Java or WebSphere Application Server, learn about WebSphere products through a newcomers page, tutorials, technology previews, training, and Redbooks, get answers to questions about WebSphere products, and join the WebSphere community, where you can keep up with the latest developments and technical papers.
- WebSphere Application Server Support page

  Take advantage of the Web-based Support and Service resources from IBM to quickly find answers to your technical questions. You can easily access this extensive Web-based support through the IBM Software Support portal at URL `http://www-3.ibm.com/software/support/` and search by product category, or by product name. For example, if you are experiencing problems specific to WebSphere Application Server, click **WebSphere Application Server** in the product list. The WebSphere Application Server Support page appears.

Chapter 5. 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. 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.
- 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 global 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.

Getting started with scripting

Scripting is a non-graphical alternative that you can use to configure and manage WebSphere Application Server. The WebSphere Application Server wsadmin tool provides the ability to run scripts. The wsadmin tool supports a full range of product administrative activities.

© Copyright IBM Corp. 2004

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

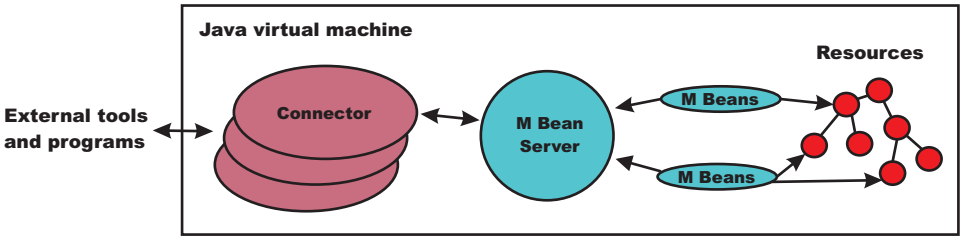

Figure 1: A WebSphere Application Server scripting solution

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.

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.

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
  - Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector
- 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:

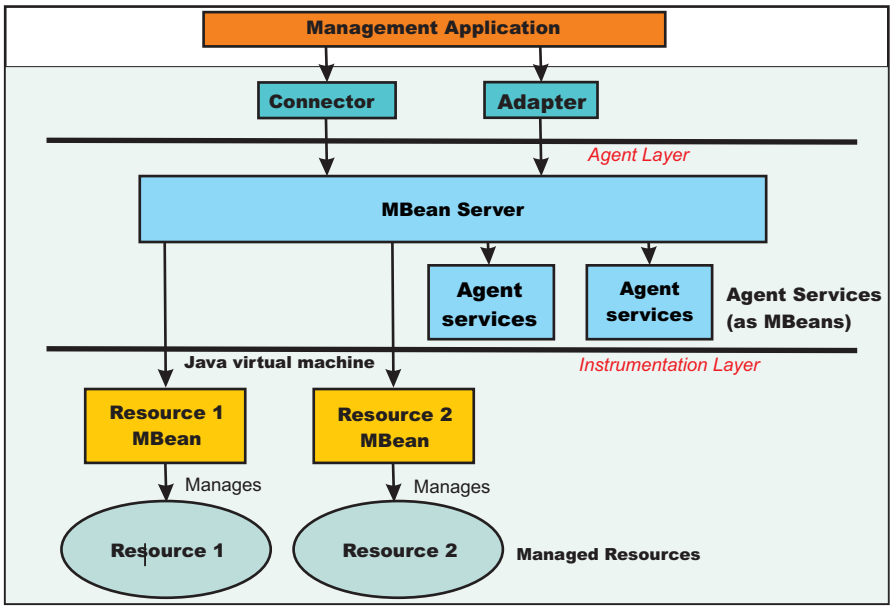

Figure 1: JMX architecture

JMX distributed administration

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

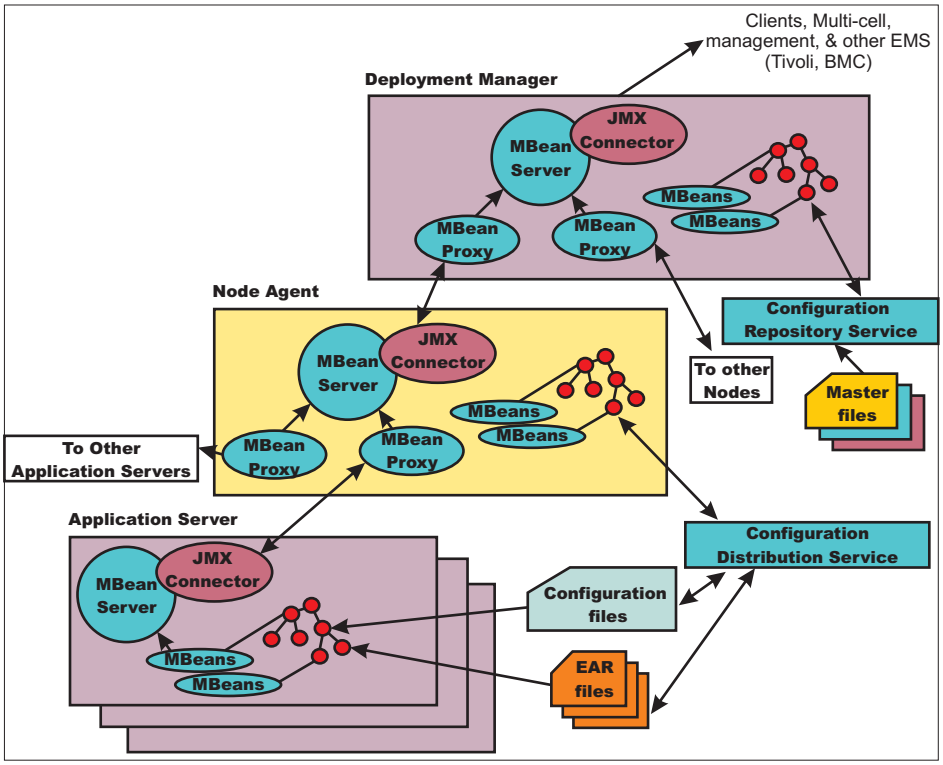

Figure 2: WebSphere Application Server distributed administration of JMX

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 have 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 an MBean:

Using Jacl:

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

Using Jacl:

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

Each WebSphere Application Server runtime MBean may have attributes, operations, and notifications. The complete documentation for each MBean 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 install directory for the product, there is a directory named web. And under that directory is another directory called mbeanDocs. In the mbeanDocs directory are several html files, one for each 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 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.
- JMX smart agents are capable of being managed through HTML browsers or by various management protocols such as Web services, Java Message Service (JMS), and Simple Network Management Protocol (SNMP).
- 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

Configuration data is stored in several XML files. The server runtime reads these files when started and responds to the component settings stored there. The configuration data includes settings for the runtime itself, such as JVM options, thread pool sizes, container setting, and port numbers the server will use. Other configuration files define J2EE resources to which the server will connect to obtain data needed by the application logic. Security settings are stored in a separate document from the server and resource configuration. Applcation-specific configuration, such as deployment target lists, session configuration, and cache settings, are stored in files under the root directory of each application.

The configuration model for WebSphere Application Server is large and complex. When viewing the XML data in the various configuration files, you can discern relationship between the configuration objects. Understanding these relationships between configuration objects is essential when writing wsadmin scripts that perform configuration functions.

Full documentation of all of the WebSphere configuration objects is available in an html table that is installed in each copy of the WebSphere product. Under the main install directory for the product, there is a directory named web. And under that directory is another directory called configDocs. In the configDocs directory are several subdirectories, one for each configuration package in the model. There is also an index.html file that ties all the individual configuration packages together in a top-level navigation tree. Each configuration package lists all the supported configuration classes. Each configuration class lists all the supported properties. Properties with names that end with the special @ character imply that property is actually a reference to some other configuration object within the configuration data. Properties with names that end with an * imply that property is actaully 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 V1.3.1. The following information is a basic summary of 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 interpreter treats the **expr** command similar to other commands and leaves the expression parsing up the **expr** command implementation. 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.

On distributed systems, special care must also be taken with path descriptions because the Jacl language uses the backslash character (\) as an escape character. To fix this, either replace each backslash with a forward slash, or use double backslashes in distributed path statements. For example: `C:/` or `C:\\`

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`. Instead, it is put into the `argv0` variable. 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 occurrences 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 occurence 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 (and discarding) 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. If an expression is true, then execute one command body, otherwise execute another command body. 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 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 servers:

```
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

The **return** command is used to return from a procedure. It is needed if you want something to return 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 ...]
}
```

For more information about Jacl, see the Scripting: Resources for Learning article.

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:

Basic function

The function is either the name of a built-in function or a Jython function. For example:

```
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    inally     for      from    global
if     importin   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 x[:-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 black slash (\) at the end of the previous line to indicate you are continuing on the next line. For example:

```
text =  "This is a tests of a long lines" \
" continuing lines here."
print text
=> This is a tests 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 `boject` 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):
  try:
     res = []
     for x in seq1:
```

```
    if x in seq2:
      res.append(x)
    return res
  except:
```

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`. The name of the program, or script, is not part of sys.argv. sys.argv is an array, so you use the index command to extract items from the argument list, for example:

```
import sys
first  =  sys.argv[0]
second  = sys.argv[1]
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 test3
 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 an while 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 running a loop 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_type, sys.exc_value

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...'
```

For more information about the Jython language, see the Scripting: Resources for Learning article.

Scripting objects

The wsadmin tool operates on configurations and running objects through the following set of management objects: AdminConfig, AdminControl, AdminApp, AdminTask, and Help. Each of these objects has commands that you can use to perform administrative tasks. To use the scripting objects, specify the scripting object, a command, and command parameters. For example:

Using Jacl:

```
$AdminConfig attributes ApplicationServer
```

Using Jython:

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

where `AdminConfig` is the scripting object, `attributes` is the command, and `ApplicationServer` is the command parameter.

To find out more specific information about each of the scripting objects, including command and command parameter information, see AdminConfig, AdminApp, AdminControl, AdminTask, or Help.

WebSphere Application Server system management separates administrative functions into two categories: functions that work with the configuration of WebSphere Application Server installations, and functions that work with the currently running objects in WebSphere 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. You can create, query, change, or remove this configuration without starting an application server process. The **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** handle configuration functionality. You can invoke configuration functions with or without being connected to a server.

The second entity represents 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 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 WebSphere Application Server scripting support provides functions to locate configuration objects, and running objects. Objects in the configuration do not always represent objects that are currently running. The **AdminControl object** manages running objects.

You can 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()
```

AdminApp object for scripted administration

Use the AdminApp object to manage applications. This object communicates with the WebSphere Application Server run time application management object to make application inquires and changes, for example:

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

Since applications are part of configuration data, any changes that you make to an application is 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.

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 or you can also use the **Help** command, for example:

Using Jacl:

```
$AdminApp help
```

Using Jython:

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

Listing applications with the wsadmin tool:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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

- Using Jacl:

  ```
  $AdminApp list
  ```

- Using Jython:

  ```
  AdminApp.list()
  ```

where:

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

Example output:

```
DefaultApplication
SampleApp
app1serv2
```

Editing application configurations with the wsadmin tool:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

1. 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 allowing application objects 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 allowing application objects 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.

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

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. 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.

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. 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. When connected to an application server, only MBeans running in that application server are visible.

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 or 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: 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: In the WebSphere Application Server, MBeans represent running objects. You can interrogate the MBean server to see the objects it contains. Use the AdminControl object to interact with running MBeans.

- 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:

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

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.

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:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use the Help object **attributes** or **operations** commands 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 obtained in step 2

3. Use the **operations** command to find out the operations 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 obtained in step number 2
operationName	(optional) is the specified operation for which you want to obtain detailed information

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 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 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 will be 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 server process
`invoke`	is an AdminControl command
`s1`	is the ID of the server specified in step number 3
`stop`	is an operation to be invoked on the server

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:

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

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 running in a WebSphere 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, etc.

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 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, etc.

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 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:

This article only applies to network deployment installations. A node synchronization is necessary in order to propagate configuration changes to the affected node or nodes. By default this occurs periodically, as long as the node can communicate with the deployment manager. It is possible to cause this to happen explicitly by performing the following steps:

1. Set the variable for node synchronize.
   - 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 server process
`completeObjectName`	is an AdminControl command
`type=NodeSync,node=`*`myNodeName`*	is 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 SyncNode object for the node *myNodeName*, 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, etc.

Example output:

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

2. Synchronize 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 running in a WebSphere server process
`invoke`	is an AdminControl command
`Sync1`	evaluates to the ID of the server specified in step number 7
`sync`	is an attribute of modify objects

Example output:

```
true
```

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

When the synchronization is complete, the files created in the `c:/WebSphere/DeploymentManager/config` directory now exists on the *`mynode`* node in the `c:/WebSphere/AppServer/config` directory.

AdminConfig object for scripted administration

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

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 connnected 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:

- Identify the configuration type and the corresponding attributes.
- Query an existing configuration object to obtain a configuration ID to use.
- Modify the existing configuration object or create a one.
- Save the configuration.

To see a list of all available commands for the AdminConfig object, see the Commands for the AdminConfig object article or 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:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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.

1. 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 representing the WebSphere Application Server configuration

listTemplates	is an AdminConfig command
JDBCProvider	is an object type

2. 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 representing 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

3. Specify the template that you want to use:
   - Using Jacl:
     ```
     set t1 [$AdminConfig listTemplates JDBCProvider "DB2 JDBC Provider (XA)"]
     ```
   - Using Jython:
     ```
     t1 = AdminConfig.listTemplates('JDBCProvider', 'DB2 JDBC Provider (XA)')
     ```

   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 representing 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.
4. Create the object with the following command:
   - Using Jacl:

```
$AdminConfig createUsingTemplate JDBCProvider $n1 {{name newdriver}} $t1
```

- 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 representing the WebSphere Application Server configuration
createUsingTemplate	is an AdminConfig command
JDBCProvider	is an object type
n1	evaluates the ID of the host node 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 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.

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Interpreting the output of the AdminConfig attributes command using scripting:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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"
```

  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"
```

  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)"
```

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"
  ```

  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*"
  ```

  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@"
  ```

  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)"
  ```

  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:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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

1. Obtain the configuration ID with 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 **Note:** The name of the object type that you input here is the one based on the XML configuration files and does not have to be the same name that the administrative console displays.
name	is the optional name of the object

  You can specify multiple `/type:name/` in the string, for example, `/type:name/type:name/type:name/`. If you just specify the type in the containment path without the name, include the colon, for example, `/type:/`. The containment path must be a path containing the correct hierarchical order. For example, if you specify `/Server:server1/Node:node/` as the containment path, you will not receive a valid configuration ID because `Node` is parent of `Server` and should come 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 resided in mynode, use the following example:
  - Using Jacl:

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

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

  To look for server1 configuration ID resided in mynode, use 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 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 representing the WebSphere Application Server configuration
`list`	is an AdminConfig command
`type`	is the object type **Note:** The name of the object type that you input here is the one based on the XML configuration files and does not have to be the same name that the administrative console displays.
scopeId	is the configuration ID of a cell, node, or server object

This command returns a list of configuration object IDs of a given type. If you specify the *scopeId*, the list of objects returned is within the scope specified. The list returned 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 *mynode*, use 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 there are more than more configuration IDs returned from the **getid** or **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 then perform pattern matching to look for a specified configuration object, refer to the Jacl syntax.

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:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 representing 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 representing 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 representing the WebSphere Application Server configuration
`defaults`	is an AdminConfig command
`type`	is an object type

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

   To get a list of attributes with 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:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 representing 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 representing the WebSphere Application Server configuration
`show`	is an AdminConfig command
`jdbcProvider1`	evaluates to the ID of host node specified in step number 2

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 representing the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`jdbcProvider1`	evaluates to the ID of host node specified in step number 3
`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. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Removing configuration objects with the wsadmin tool:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use this task to delete a configuration object from the configuration repository. This action only affects the configuration. If there is a running instance of a configuration object when you remove the configuration, the change has no effect on the running instance.

1. Assign the ID string that identifies the server 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 representing 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

2. 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 representing the WebSphere Application Server configuration
`remove`	is an AdminConfig command
`s1`	evaluates the ID of the server specified in step number 2

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

The WebSphere Application Server configuration no longer contains a specific server object. Running servers are not affected.

Changing the WebSphere Application Server configuration using wsadmin:

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information. For this task, the wsadmin scripting client must be connected to the deployment manager server in a network deployment environment.

You can use the wsadmin AdminConfig and AdminApp objects to make changes to the WebSphere Application Server configuration. The purpose of this article is to illustrate the relationship between the commands used to change the configuration and the files used to hold configuration data. This discussion assumes that you have a network deployment installation, but the concepts are very similar for a WebSphere Application Server installation.

1. 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 representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is the object type
mynode	is the name of the object that will be modified

2. 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 representing the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Server`	is an AdminConfig object
`n1`	evaluates to the ID of host node specified in step number 2
`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, only 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.

3. 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 representing the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`serv1`	evaluates to the ID of host node specified in step number 3
`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.
4. Install an application on the server.
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Modifying nested attributes with the wsadmin tool:

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.

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:

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:

- Using Jacl:

  ```
  $AdminConfig save
  ```

- 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.

You can use the **reset** command of the AdminConfig object to undo changes that you made to your configuration since your last save.

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. 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 you install. You can use the AdminTask object commands to access these commands.

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. Depending on the administrative command, it 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.

Obtaining online help using scripting:

There are three levels of online help available with the administrative commands. The top level help provides general information for the AdminTask object and the commands associated with it. 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, its parameters, and steps, if any. 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 a * 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 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 adminstrative 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.
  ```

- Obtaining help about an administrative command:

  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 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 only contains the description of the target object. 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, a * will be 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:

- Target object area requires the configuration object name of a J2CResourceAdapter to be provided.
- 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 execute an admin command in batch mode, for example:

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

  See “Administrative command invocation syntax” on page 508 for more information about specifying argument options.
- There is no step associated with this administrative command.

- Obtain help on a command step.

  Step specific help provides the following:
  - 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, a * exists in front of the name.

  The following example obtain 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:
  - It does not support collection. Only one set of parameter values for the clusterName and perferLocal parameters is allowed.
  - It contains two input arguments with one argument indicated as required. The required arguments is clusterName and the non-required parameter is perferLocal. 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 execute 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 508 for more information about specifying parameter options.

Invoking an administrative command in batch mode:

Perform the following steps to invoke an administrative command in batch mode. To invoke an administrative command in interactive mode, see “Invoking an administrative command in interactive mode” on page 94.

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 being invoked

   - 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 allowing administrative command management
commandName	is the name of the administrative command being invoked

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 allowing administrative command management
commandName	is the name of the administrative command being invoked

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 listDataSource: `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. There are three required options 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 {} in Jacl and a pair of [] 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 allowing administrative command management
commandName	is the name of the administrative command being invoked

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 J2CResourceAdapter.

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 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. There are three required options for this administrative command. The required options are -connectionFactoryInterface, -name, and -jndiName. There are two optional options. They are -description and -authDataAlias. In addition, the -interactive option is available. Options are specified in the option string which is enclosed by a pair of {} in Jacl and a pair of [] in Jython.

- 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 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:

Perform the following steps to invoke an administrative command in interactive mode. To invoke an administrative command in batch mode, see “Invoking an administrative command in batch mode” on page 89.

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 allowing administrative command management
commandName	is the name of the administrative command being invoked
`-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 allowing administrative command management
commandName	is the name of the administrative command being invoked
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 allowing administrative command management

commandName	is the name of the administrative command being invoked
-interactive	is the interactive option
commandOptions	is the command option 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 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 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` In this example, there are five options available: • -connectionFactoryInterface • -name • -jndiName • -description • -authDataAlias Each requires an option value. Only three of the options are required and are denoted with a `*`.

- 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 allowing administrative command management
commandName	is the name of the administrative command being invoked
targetObject	is the target object string for the invoked admin command. The expect target object varies with each admin command. Consult the on-line help on the invoked admin command to learn more about what to specify as target object.
`-interactive`	is the interactive option

commandOptions	is the command option 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 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 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` In this example, there are five options available: • -connectionFactoryInterface • -name • -jndiName • -description • -authDataAlias Each requires an option value. Only three of the options are required and are denoted with a `*`.

- 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 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 3-5.
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 clicking Enter. 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 adminstative 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 on-line help for the command. Not all menu choices are available. Previous is not available if current step is the first step. Next is not be available if 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 there are still steps missing required inputs. Default selection is `F` (Finish) if there is no missing required input in any step.

For commands with steps, this is the menu to exit interactive mode by either canceling or executing 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 with each having the same set of parameters. If a step supporting 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 so as 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 allows 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 “*” notation is used to denote required parameter. 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. For each writable parameter, you can accept the existing value by pressing the Enter key. To add or change an existing value, user simply enters a new value and then presses Enter. For a read-only parameter, the parameter and its value are displayed. However, user is not given the prompt to modify its value. Once user steps through all the parameters, wsadmin leads user back to the collection step menu.

Collect user inputs for non-collection step

There are two parts for this step. The first part is to display the current or default parameter values for the selected step as shown below:

```
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]
```

There is no prompting in this part. Instead, this part is more like a help 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 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`. Since 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 should always accept the default selection to continue to the next part. Otherwise, if there is no data in the selected step, selecting Cancel will 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 clicking 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.

Starting the wsadmin scripting client

The WebSphere Application Server wsadmin tool provides the ability to run scripts. You can use the wsadmin tool to manage a WebSphere Application Server V6.0 installation, as well as configuration, application deployment, and server run-time operations. The WebSphere Application Server only supports the Jacl and Jython scripting languages.

The wsadmin launcher makes several WebSphere Application Server scripting objects available: AdminConfig, AdminControl, AdminApp, AdminTask, and Help. Scripts use these objects for application management, configuration, operational control, and for communication with MBeans that run in WebSphere Application Server processes.

You must start the wsadmin scripting client before you perform any other task using scripting.

1. Locate the command that starts the wsadmin scripting client.

   The command for invoking a scripting process is located in the *install_root*/profiles/*profile_name*/bin directory. Use the `wsadmin.bat` file for a Windows system, and the `wsadmin.sh` file for a Linux or a UNIX system.

2. Start the wsadmin scripting client. 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:

Option for starting the wsadmin scripting client:	Explanation:	Examples:
Run scripting commands interactively	Run wsadmin with an option other than `-f` or `-c` or without an option. An interactive shell is displayed 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.	Using Jacl on Windows systems: `wsadmin.bat` Using Jacl on Unix systems: `wsadmin.sh` If security is enabled: `wsadmin.sh -user wsadmin -password wsadmin` Using Jython on Windows systems: `wsadmin.bat -lang jython` Using Jython on Unix systems: `wsadmin.sh -lang jython` By default security is enabled: `wsadmin.sh -lang jython -user wsadmin -password wsadmin` Example output: `WASX7209I: Connected to process server1 on node myhost using SOAP connector; The type of process is: UnManagedProcess` `WASX7029I: For help, enter: "$Help help"` `wsadmin>$AdminApp list` `adminconsole` `DefaultApplication` `ivtApp` `wsadmin>exit`

Run scripting commands as individual commands	Run the wsadmin tool with the -c option.	Using Jacl on Windows systems: `wsadmin -c "$AdminApp list"` Using Jacl on Unix systems: `wsadmin.sh -c "\$AdminApp list"` or `wsadmin.sh -c '$AdminApp list'` Using Jython on Windows systems: `wsadmin -lang jython -c "AdminApp.list()"` Using Jython on Linux or Unix systems: `wsadmin.sh -lang jython -c 'AdminApp.list()'` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector;  The type of process is: UnManagedProcess adminconsole DefaultApplication ivtApp`
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.	Using Jacl on Windows systems: `wsadmin -f al.jacl` Using Jacl on Unix systems: `wsadmin.sh -f al.jacl` where the `al.jacl` file contains the following commands: `set apps [$AdminApp list] puts $apps` Using Jython on Windows systems: `wsadmin -lang jython -f  al.py` Using Jython on Unix systems: `wsadmin.sh -lang jython -f  al.py` where the `al.py` file contains the following commands: `apps = AdminApp.list() print apps` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector; The  type of process is: UnManagedProcess adminconsole DefaultApplication ivtApp`

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. 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.	Using Jacl on Windows systems: `wsadmin.bat -profile alprof.jacl` Using Jacl on Linux or Unix systems: `wsadmin.sh -profile alprof.jacl` where the `alprof.jacl` file contains the following commands: `set apps [$AdminApp list]` `puts "Applications currently installed:\n$apps"` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector;  The type of process is: UnManagedProcess` `Applications currently installed:` ` adminconsole` ` DefaultApplication` ` ivtApp` ` WASX7029I: For help, enter: "$Help help"` ` wsadmin>` Using Jython on Windows systems: `wsadmin.bat -lang jython -profile alprof.py` Using Jython on Linux or Unix systems: `wsadmin.sh -lang jython -profile alprof.py` where the `alprof.py` file contains the following commands: `apps = AdminApp.list()` `print "Applications currently installed:\n " + apps` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector; The type of process is: UnManagedProcess` `Applications currently installed:` ` adminconsole` ` DefaultApplication` ` ivtApp` ` WASX7029I: For help, enter: "Help.help()"` ` wsadmin>`

Scripting: Resources for learning

Use the following links to find relevant supplemental information about the Jacl and Jython scripting languages, and about using scripting with WebSphere Application Server. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links

are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming instructions and examples
- Java command language
- Jacl: A Tcl implementation in Java
- Charming Jython
- Jython
- Sample scripts for WebSphere Application Server

Deploying applications using scripting

This topic contains the following tasks:
- Installing applications
- Uninstalling applications

Installing applications with the wsadmin tool

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

On a single server installation, the server must be running before you install an application. See the “Starting servers using scripting” on page 157 article for more information. On a network deployment installation, the deployment manager must be running before you install an application. See the “startManager command” on page 560article for more information.

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 a set of options. See the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 307 article for a list of valid options for the **install** and **installinteractive** commands. 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:

```
AdminApp.options()
```

For more information for the **options**, **install**, or **installinteractive** commands, see the “Commands for the AdminApp object” on page 282 article.

The application that you install must be an enterprise archive file (EAR), a Web archive (WAR) file, or a Java archive (JAR) file. The archive file must end in `.ear`, `.jar` or `.war` for the wsadmin tool to be able to install it. The wsadmin tool uses these extensions to figure out the archive type. If the file is a WAR or JAR file, it will be automatically wrapped as an EAR file.

If you are installing an application that has the AdminApp useMetaDataFromBinary option specified, then you can only install this application on a WebSphere Application Server V6.x deployment target. This also applies to editing the application, using the AdminApp **edit** command, after you install it. If you use the V5.x wsadmin tool to install or edit an application on a WebSphere Application Server V6.x cell, only the steps available for the V5.x wsadmin tool will be shown.

Perform the following steps to install an application into the run time:

1. Install the application.
   - Using batch mode:
     - For a single server installation only, the following example uses the EAR file and the command option information to install the application:
       - Using Jacl:

         ```
         $AdminApp install c:/MyStuff/application1.ear {-server serv2}
         ```

       - Using Jython list:

         ```
         AdminApp.install('c:/MyStuff/application1.ear', ['-server', 'serv2'])
         ```

       - Using Jython string:

         ```
         AdminApp.install('c:/MyStuff/application1.ear', '[-server serv2]')
         ```

       where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object supporting application object management
`install`	is an AdminApp command
MyStuff/application1.ear	is the name of the application to install
`server`	is an installation option
serv2	is the value of the server option

     - 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 Jacl:

         ```
         $AdminApp install c:/MyStuff/application1.ear {-cluster cluster1}
         ```

       - Using Jython list:

         ```
         AdminApp.install('c:/MyStuff/application1.ear', ['-cluster', 'cluster1'])
         ```

       - Using Jython string:

         ```
         AdminApp.install('c:/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

   - Using interactive mode, the following command changes the application information by prompting you through a series of installation tasks:
     - Using Jacl:

       ```
       $AdminApp installInteractive c:/MyStuff/application1.ear
       ```

     - Using Jython:

       ```
       AdminApp.installInteractive('c:/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

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Uninstalling applications with the wsadmin tool

Before starting this task, the wsadmin tool must be running. See “Starting the wsadmin scripting client” on page 103 for more information.

Steps to uninstall an application follow:

1. 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

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Uninstalling an application removes it from the WebSphere Application Server configuration and from all the servers that the application was installed on. The application binaries (EAR file contents) are deleted from the installation directory. This occurs when the configuration is saved for single server WebSphere Application Server editions or when the configuration changes are synchronized from deployment manager to the individual nodes for network deployment configurations.

Managing deployed applications using scripting

This topic contains the following tasks:

Starting applications with scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

Before starting an application, it must be installed. See the "Installing applications with the wsadmin tool" on page 107 article for more information.

Perform the following steps to start an application:

1. 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 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
```

2. 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 server process
`invoke`	is an AdminControl command
`appManager`	evaluates to the ID of the server specified in step number 1
`startApplication`	is an attribute of modify objects
myApplication	is the value of the startApplication attribute

Updating installed applications with the wsadmin tool

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Before starting an application, it must be installed. See the “Installing applications with the wsadmin tool” on page 107 article for more information.

Both the **update** command and the **updateinteractive** command support a set of options. See the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 307 article for a list of valid options for the **update** and **updateinteractive** commands. 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()
```

For more information for the **options**, **update**, or **updateinteractive** commands, see the “Commands for the AdminApp object” on page 282 article. Perform the following steps to update an application:

1. 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
       c:/apps/app1/my.xml -contenturi app1.jar/my.xml}
       ```

     - Using Jython string:

       ```
       AdminApp.update('app1', 'file', '[-operation update -contents
       c:/apps/app1/my.xml -contenturi app1.jar/my.xml]')
       ```

– Using Jython list:

```
AdminApp.update('app1', 'file', ['-operation', 'update', '-contents',
'c:/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 supporting 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 will be updated.
  – Using Jacl:

```
$AdminApp update app1 modulefile {-operation addupdate -contents
c:/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb
-BindJndiForEJBNonMessageBinding {{"Increment Enterprise Java Bean"
 Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
```

  – Using Jython string:

```
AdminApp.update('app1', 'modulefile', '[-operation addupdate -contents
c:/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb
-BindJndiForEJBNonMessageBinding [["Increment Enterprise Java Bean
" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

  – Using Jython list:

```
bindJndiForEJBValue = [["Increment Enterprise Java Bean",
"Increment", " Increment.jar,META-INF/ejb-jar.xml", "Inc"]]

AdminApp.update('app1', 'modulefile', ['-operation', 'addupdate', '-contents',
'c:/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 supporting 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 Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option
bindJndiForEJBValue	is a Jython variable containing 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 c:/apps/app1/app1Partial.zip}
    ```
  - Using Jython string:

    ```
    AdminApp.update('app1', 'partialapp', '[-contents c:/apps/app1/app1Partial.zip]')
    ```
  - Using Jython list:

    ```
    AdminApp.update('app1', 'partialapp', ['-contents', 'c:/apps/app1/app1Partial.zip'])
    ```

  where:

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object supporting 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

- The following command updates the entire deployed application:
  - Using Jacl:

    ```
    $AdminApp update app1 app {-operation update -contents
    c:/apps/app1/newApp1.jar -usedefaultbindings -nodeployejb
    -BindJndiForEJBNonMessageBinding {{"Increment Enterprise
    Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
    ```
  - Using Jython string:

    ```
    AdminApp.update('app1', 'app', '[-operation update -contents
    c:/apps/app1/newApp1.ear -usedefaultbindings -nodeployejb
    -BindJndiForEJBNonMessageBinding [["Increment Enterprise
    Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
    ```
  - Using Jython list:

    ```
    bindJndiForEJBValue = [["Increment Enterprise Java Bean",
    "Increment", " Increment.jar,META-INF/ejb-jar.xml", "Inc"]]

    AdminApp.update('app1', 'app', ['-operation', 'update', '-contents',
    'c:/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 supporting 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 Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option
bindJndiForEJBValue	is a Jython variable containing the value of the BindJndiForEJBNonMessageBinding option

2. Save the configuration changes. See the "Saving configuration changes with the wsadmin tool" on page 84 article for more information.
3. In a network deployment environment only, synchronize the node. See the "Synchronizing nodes with the wsadmin tool" on page 69 article for more information.

Stopping applications with scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

The following example stops all running applications on a server:

1. 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
```

2. 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
   ```

3. 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.

Once you complete the steps for this task, all running applications on the server are stopped.

Listing the modules in an installed application with scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use the AdminApp object **listModules** command to list the modules in an installed application. For example:

- 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 supporting 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

The following example lists all modules on all enterprise applications installed on `server1` in `node1`:

Note: * means that the module is installed on server1 node node1 and other node and/or server.

+ means that the module is installed on server1 node node1 only means that the module is not installed on server1 node node1.

```
 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 application state using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

The following example queries for the presence of 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 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

If *myApplication* is running, then there should be an MBean created for it. Otherwise, the command returns nothing. If *myApplication* is running, the following is the example output:

```
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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

The following example uses the AdminConfig object to disable application loading in deployed targets:

1. 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 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)
   ```

2. 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 representing the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates to the ID of the deployment object specified in step number 1
`deployedObject`	is an attribute

`targetMap1`	a variable name
`targetMappings`	is an attribute
`lindex`	a Jacl command
`print`	a Jython command

3. 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"]])
     ```

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring applications for session management using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object. The following example uses the AdminConfig object to configure session manager for the application.

1. 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 deployment
     ```

   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)
   ```

2. Retrieve the applicaton deployment 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 representing the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates to the ID of the deployment object specified in step number 1
`deployedObject`	is an attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|
deployment.xml#ApplicationDeployment_1)
```

3. To obtain a list of attributes you can set for 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 representing 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"
```

4. Set up the attributes for the session manager. The following example sets three top level attributes in the session manager. You can modify the example to set other attributes of 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 the AdminConfig object.

- Using Jacl:

```
set attr1 [list enableSecurityIntegration true]
set attr2 [list maxWaitTime 30]
set attr3 [list sessionPersistenceMode NONE]
set attrs [list $attr1 $attr2 $attr3]
set sessionMgr [list sessionManagement $attrs]
```

  Example output using Jacl:

```
sessionManagement {{enableSecurityIntegration true}
{maxWaitTime 30} {sessionPersistenceMode NONE}}
```

- Using Jython:

```
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
attrs = [attr1, attr2, attr3]
sessionMgr = [['sessionManagement', attrs]]
```

  Example output using Jython:

```
[[sessionManagement, [[enableSecurityIntegration, true],
[maxWaitTime, 30], [sessionPersistenceMode, NONE]]]
```

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

5. 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 representing the WebSphere Application Server configuration
`create`	is an AdminConfig command
`ApplicationConfig`	is an attribute
`appDeploy`	evaluates to the ID of the deployed application specified in step number 2
`list`	is a Jacl command

`sessionMgr`	evaluates to the ID of the session manager specified in step number 4

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/
myApp|deployment.xml#ApplicationConfig_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring applications for session management in Web modules using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object. This example uses the AdminConfig object to configure session manager for Web module in the application.

1. 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)
   ```

2. Get all the modules in the application and assign it 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 representing the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates to the ID of the deployment object specified in step number 1
`deployedObject`	is an attribute

3. To obtain a list of attributes you can set for 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 representing 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"
```

4. 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]]
```

5. Set up the attributes for 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

6. Create the session manager for each Web module in the application. You can modify the following example to set other attributes of session manager in Web module configuration.
   - Using Jacl:

     ```
     foreach module $mod1 {
          if ([regexp WebModuleDeployment $module] == 1} {
             $AdminConfig create WebModuleConfig $module $webAttrs
       $AdminConfig save
          }
      }
     ```

   - Using Jython:

```
arrayModules = mod1[1:len(mod1)-1].split(" ")
for module in arrayModules:
 if module.find('WebModuleDeployment') != -1:
  AdminConfig.create('WebModuleConfig', module, webAttrs)
  Adminconfig.save()
```

Example output:

```
myWebModuleConfig(cells/mycell/applications/myApp.ear/
deployments/myApp|deployment.xml#WebModuleConfiguration_1)
```

7. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
8. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Exporting applications enables you to back them up and preserve their binding information.

- Export an enterprise application to a location of your choice, for example:
  - Using Jacl:

    ```
    $AdminApp export app1 C:/mystuff/exported.ear
    ```

  - Using Jython:

    ```
    AdminApp.export('app1', 'C:/mystuff/exported.ear')
    ```

  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:

    ```
    $AdminApp exportDDL app1 C:/mystuff
    ```

  - Using Jython:

    ```
    AdminApp.exportDDL('app1', 'C:/mystuff')
    ```

  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

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

Perform the following steps to configure an application server to use a shared library.

1. 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 representing 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)
   ```
2. Create the shared library in the server. For example:
   - Using Jacl:

     ```
     $AdminConfig create Library $serv {{name mySharedLibrary}
     {classPath c:/mySharedLibraryClasspath}}
     ```
   - Using Jython:

     ```
     print AdminConfig.create('Library', serv, [['name', 'mySharedLibrary'],
     ['classPath',  'c:/mySharedLibraryClasspath']])
     ```

   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
`Library`	is an attribute

`serv`	evaluates the ID of the server 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)
```

3. 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 representing the WebSphere Application Server configuration
`list`	is an AdminConfig command
`ApplicationServer`	is an attribute
`serv`	evaluates the ID of the server specified in step number 1
`print`	is a Jython command

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#ApplicationServer_1
```

4. Identify the class loader in the application server and assign it to the classLoader variable. For example:
   - To use the existing class loader 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 representing the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
appServer	evaluates the ID of the application server specified in step number 3
classloaders	is an attribute
print	is a Jython command

- To create a new class loader, do the following step:
  - 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 representing the WebSphere Application Server configuration
create	is an AdminConfig command
Classloader	is an attribute
appServer	evaluates the ID of the application server 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)
  ```

5. Associate the created shared library with the application server through the class loader. For example:
   - Using Jacl:

     ```
     $AdminConfig create LibraryRef $classLoader1
     {{libraryName MyshareLibrary} {sharedClassloader true}}
     ```

   - Using Jython:

     ```
     print AdminConfig.create('LibraryRef', classLoader1,
     [['libraryName', 'MyshareLibrary'],  ['sharedClassloader', 'true']])
     ```

   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 attribute

`classLoader1`	evaluates the ID of the class loader specified in step number 4
`libraryName`	is an attribute
`MyshareLibrary`	is the value of the attribute
`sharedClassloader`	is an attribute
`true`	is the value of the attribute
`print`	is a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#LibraryRef_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring a shared library for an application using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

You can use the AdminApp object to set certain configurations in the application. This example uses the AdminConfig object to configure a shared library for an application.

1. 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:

```
set library [$AdminConfig create Library $n1 {{name mySharedLibrary}
{classPath c:/mySharedLibraryClasspath}}]
```

– Using Jython:

```
library = AdminConfig.create('Library', n1, [['name', 'mySharedLibrary'],
['classPath', 'c:/mySharedLibraryClasspath']])
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
`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)
```

2. 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)
```

3. 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)
   ```

4. 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)
```

5. Associate the shared library in the application through the class loader. For example:
   - Using Jacl:

     ```
     $AdminConfig create LibraryRef $classLoad1 {{libraryName
     MyshareLibrary} {sharedClassloader true}}
     ```

   - Using Jython:

     ```
     print AdminConfig.create('LibraryRef', classLoad1, [['libraryName',
     'MyshareLibrary'], ['sharedClassloader', 'true']])
     ```

   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
sharedClassloader	is an attribute
true	is the value of the sharedClassloader attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#LibraryRef_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Setting background applications using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to enable or disable a background application. 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. 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 representing 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

2. 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 representing 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

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring servers with scripting

This topic contains the following tasks:

- “Configuring a Performance Manager Infrastructure service using scripting” on page 143
- “Limiting the growth of Java virtual machine log files using scripting” on page 145
- “Configuring an ORB service using scripting” on page 146
- “Configuring for processes using scripting” on page 148
- “Configuring transaction properties for a server using scripting” on page 149
- “Setting port numbers kept in the serverindex.xml file using scripting” on page 151
- “Disabling components using scripting” on page 152
- “Disabling services using scripting” on page 153
- “Dynamic caching with scripting” on page 154

Creating a server using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Creating an application server involves a configuration command. Perform the following steps to create a server:

1. Obtain the configuration ID of the object and assign it to the `node` variable, for example:

   Using Jacl:

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

   Using Jython:

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

2. Create the server using the node that you specified in the first step:

   Using Jacl:

   ```
   $AdminConfig create Server $node {{name myserv}
   {outputStreamRedirect {{fileName myfile.out}}}}
   ```

   Using Jython:

   ```
   AdminConfig.create('Server', node, [['name', 'myserv'],
   ['outputStreamRedirect', [['fileName', 'myfile.out']]]])
   ```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring the Java virtual machine using scripting

An example modifying the Java virtual machine (JVM) of a server to turn on debug follows:

- Identify the server and assign it to the server1 variable.

  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)
  ```

- Identify the JVM belonging to this server and assign it to the jvm variable.

  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)
```

- Modify the JVM to turn on debug.

  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"]])
  ```

- Save the changes with the following command:

  Using Jacl:

  ```
  $AdminConfig save
  ```

  Using Jython:

  ```
  AdminConfig.save()
  ```

Configuring enterprise bean containers using scripting

Before starting this task, the wsadmin tool must be running. See “Starting the wsadmin scripting client” on page 103 for more information.

Perform the following steps to configure an enterprise bean container:

1. Identify the application server and assign it to the serv1 variable. For example:
   - 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
     ```

   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 representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`/Cell:`*`mycell`*`/Node:`*`mynode`*`/Server:`*`server1`*`/`	is the hierarchical containment path of the configuration object
`Cell`	is the object type
`mycell`	is the optional name of the object
`Node`	is the object type
`mynode`	is the optional name of the object
`Server`	is the object type
`server1`	is the optional name of the object

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

2. Identify the EJB container belonging to the server and assign it to the ejbc1 variable. For example:
   - Using Jacl:

     ```
     set ejbc1 [$AdminConfig list EJBContainer $serv1]
     ```

   - Using Jython:

     ```
     ejbc1 = AdminConfig.list('EJBContainer', serv1)
     print ejbc1
     ```

   where:

`set`	is a Jacl command
`ejbc1`	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
`list`	is an AdminConfig command
`EJBContainer`	is the object type **Note:** The name of the object type that you input here 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 specified in step number 1

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
   ```

3. View all the attributes of the enterprise bean container.
   - The following example command does not show 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)]
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`print`	is a Jython command
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`showall`	is an AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container specified in step number 2

- The following command example includes 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)]]]
```

where:

$	is a Jacl operator for substituting a variable name with its value
print	is a Jython command
AdminConfig	is an object representing the WebSphere Application Server configuration
showall	is an AdminConfig command
ejbc1	evaluates to the ID of the enterprise bean container specified in step number 2

4. 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]]]])
       ```

     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
ejbc1	evaluates to the ID of the enterprise bean container specified in step number 2
cacheSettings	is an attribute of modify objects
cacheSize	is an attribute of modify objects
2500	is the value of the cacheSize attribute
cleanupInterval	is an attribute of modify objects
3500	is 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]])
       ```

     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
`ejbc1`	evaluates to the ID of the enterprise bean container specified in step number 2
`inactivePoolCleanupInterval`	is an attribute of modify objects
`15000`	is the value of the inactivePoolCleanupInterval attribute

5. Save the changes. For example:
   - Using Jacl:

     ```
     $AdminConfig save
     ```
   - 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

Configuring a Performance Manager Infrastructure service using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use the following steps to configure the Performance Manager Infrastructure (PMI) service for an application server:

1. Identify the application server and assign it to the s1 variable, for example:
   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```
   - Using Jython:

     ```
     s1 = AdminConfig.getid('Cell:mycell/Node:mynode/Server:server1/')
     ```

   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 an attribute
`mycell`	is the value of the Cell attribute
`Node`	is an attribute
`mynode`	is the value of the Node attribute
`Server`	is an attribute
`server1`	is the value of the Server attribute

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

2. Identify the PMI service that belongs to the server and assign it to the `pmi` variable, for example:
   - Using Jacl:

     ```
     set pmi [$AdminConfig list PMIService $s1]
     ```

   - Using Jython:

     ```
     pmi = AdminConfig.list('PMIService', s1)
     print pmi
     ```

   where:

`set`	is a Jacl command
`pmi`	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
`list`	is an AdminConfig command
`PMIService`	is an AdminConfig object
`s1`	evaluates to the ID of the application server specified in step number 1

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
   ```

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

     ```
     $AdminConfig modify $pmi {{enable true}
     {initialSpecLevel beanModule=H:cacheModule=H:connectionPoolModule=
     H:j2cModule=H:jvmRuntimeModule=H:orbPerfModule=H:servletSessionsModule
     =H:systemModule=H:threadPoolModule=H:transactionModule=H:
     webAppModule=H:webServicesModule=H:wlmModule=H:wsgwModule=H}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(pmi, [['enable', 'true'],
     ['initialSpecLevel', 'beanModule=H:cacheModule=H:connectionPoolModule=
     H:j2cModule=H:jvmRuntimeModule=H:orbPerfModule=H:servletSessionsModule=
     H:systemModule=H:threadPoolModule=H:transactionModule=H:webAppModule=H:
     webServicesModule=H:wlmModule=H:wsgwModule=H']])
     ```

   This example enables PMI service and sets the specification levels for all of components in the server. The following are the valid specification levels for the components:

N	represents **none**
L	represents **low**
M	represents **medium**
H	represents **high**
X	represents **maximum**

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Limiting the growth of Java virtual machine log files using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use the following example to configure the rotation policy settings for Java virtual machine (JVM) logs:

1. Identify the application server and assign it to the server1 variable, for example:
   - 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)
   ```

2. Identify the stream log and assign it to the log variable, for example:
   - 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)
   ```

3. List the current values of the stream log, for example:
   - 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}
  ```

4. 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]])
       ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring an ORB service using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to modify the Object Request Broker (ORB) service for an application server:

1. Identify the application server and assign it to the server variable:
   - 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)
```

2. Identify the ORB belonging to the server and assign it to the orb variable:
   - Using Jacl:

     ```
     set orb [$AdminConfig list ObjectRequestBroker $s1]
     ```

   - Using Jython:

     ```
     orb = AdminConfig.list('ObjectRequestBroker', s1)
     print orb
     ```

   where:

`set`	is a Jacl command
`orb`	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
`list`	is an AdminConfig command
`ObjectRequestBroker`	is an AdminConfig object
`s1`	evaluates to the ID of server specified in step number 1
`print`	a Jython command

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
   ```

3. Modify the 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']])
     ```

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
`orb`	evaluates to the ID of ORB specified in step number 2
`connectionCacheMaximum`	is an attribute
`252`	is the value of the connectionCacheMaximum attribute
`noLocalCopies`	is an attribute
`true`	is the value of the noLocalCopies attribute

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring for processes using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a process:

1. Identify the server and assign it to the s1 variable. For example:
   - 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)
   ```

2. Identify the process definition belonging to this server and assign it to the processDef variable. For example:
   - Using Jacl:

     ```
     set processDef [$AdminConfig list JavaProcessDef $s1]
     set processDef [$AdminConfig showAttribute $s1 processDefinition]
     ```

   - Using Jython:

     ```
     processDef = AdminConfig.list('JavaProcessDef', s1)
     print processDef
     processDef = AdminConfig.showAttribute(s1, 'processDefinition')
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#JavaProcessDef_1)
   ```

3. Change the attributes.
   - On distributed systems, the following example changes the working directory.
     - Using Jacl:

       ```
       $AdminConfig modify $processDef {{workingDirectory c:/temp/user1}}
       ```

     - Using Jython:

       ```
       AdminConfig.modify(processDef, [['workingDirectory', 'c:/temp/user1']])
       ```

   - The following example modifies the stderr file name:
     - 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:
     - 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]]]])
       ```

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring transaction properties for a server using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure the runtime transaction properties for an application server.

1. Identify the transaction service MBean for the application server. The following command returns the transaction service MBean for *server1*.
   - 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
```

where:

`set`	is a Jacl command
`ts`	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
`cell=mycell,node=mynode,process=server1,` `type=TransactionService`	is 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 attributes.
   - Using Jacl:

```
$AdminControl invoke $ts {{transactionLogDirectory
c:/WebSphere/AppServer/tranlog/server1}
{clientInactivityTimeout 30} {totalTranLifetimeTimeout 180}}
```

   - Using Jython:

```
AdminControl.invoke(ts, [['transactionLogDirectory',
'c:/WebSphere/AppServer/tranlog/server1'],
['clientInactivityTimeout', 30],  ['totalTranLifetimeTimeout', 180]])
```

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
`invoke`	is an AdminControl command
`ts`	evaluates to the ID of the transaction service specified in step number 1
`transactionLogDirectory`	is an attribute
c:/WebSphere/AppServer/tranlog/server1	is the value of the transactionLogDirectory attribute
`clientInactivityTimeout`	is an attribute
`30`	is the value of the clientInactivityTimeout attribute specified in seconds. A value of 0 means that there is no timeout limit.
`totalTranLifetimeTimeout`	is an attribute

180	is the value of the totalTranLifetimeTimeout attribute specified in milliseconds. A value of 0 means that there is no timeout limit.

Setting port numbers kept in the serverindex.xml file using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

This topic provides reference information about modifying port numbers in the `serverindex.xml` file. The end points of the `serverindex.xml` file are part of different objects in the configuration.

Use the following attributes to modify the end point information of the end point attributes for a process:

- **BOOTSTRAP_ADDRESS** of server1 process

  An attribute of the NameServer object that exists inside the server. It is used by the naming client to specify the naming server to look up the initial context. To modify its end point, obtain the ID of the NameServer object and issue a **modify** command, for example:
  - 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]]]])
    ```
- **SOAP_CONNECTOR-ADDRESS** of server1 process

  An attribute of the SOAPConnector object that exists inside the server. It is the port that is used by HTTP transport for incoming SOAP requests. To modify its end point, obtain the ID of the SOAPConnector object and issue a modify command, for example:
  - 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]]]])
    ```
- **DRS_CLIENT_ADDRESS** of server1 process

  An attribute of the SystemMessageServer object that exists inside the server. It is the port used 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.

  To modify the end point of the DRS_CLIENT_ADDRESS attribute, obtain the ID of the SystemMessageServer object and issue a **modify** command, for example:
  - 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]]]])
    ```
- **JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS** of server1 process

An attribute of the JMSServer object that exists inside the server. These are ports used to configure the WebSphere Application Server JMS provider topic connection factory settings. To modify its end point, 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]]]])
```

- **NODE_DISCOVERY_ADDRESS** of nodeagent process

  An attribute of the NodeAgent object that exists inside the server. It is the port used to receive the incoming process discovery messages inside a node agent process. To modify its end point, 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]]]])
```

- **CELL_DISCOVERY_ADDRESS** of dmgr process

  An attribute of the deploymentManager object that exists inside the server. It is the port used to receive the incoming process discovery messages inside a deployment manager process. To modify its end point, 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]]]])
```

Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.

In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Disabling components using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to disable the name server component of a configured server. You can modify this example to disable a different component.

1. 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)
   ```

2. List the components belonging to the server and assign them to the components variable.
   - 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)
   ```

3. Identify the name server 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 index 2.
   - 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)
   ```

4. Disable the name server component by changing the nested initialState attribute belonging to the stateManagement attribute. For example:
   - Using Jacl:

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

   - Using Jython:

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

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Disabling services using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to disable the trace service of a configured server. You can modify this example to disable a different service.

1. 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)
   ```
2. 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)
   ```
3. Identify the trace service and assign it to the traceService variable.

   Since trace service is the 7th 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)
   ```
4. 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']])
     ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Dynamic caching with scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

To see a list of parameters associated with dynamic caching, use the **attributes** command. For example:

```
$AdminConfig attributes DynamicCache
```

Perform the following steps to enable servlet caching:

1. Locate the server object. The following example selects the first server found:

   Using Jacl:

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

   Using Jython:

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

2. List the web containers and assign them to the `wc` variable, for example:

   Using Jacl:

   ```
   set wc [$AdminConfig list WebContainer $s1]
   ```

   Using Jython:

   ```
   wc = AdminConfig.list('WebContainer', s1)
   ```

3. Set the enableServletCaching attribute to `true` and assign it to the `serEnable` variable, for example:

   Using Jacl:

   ```
   set serEnable "{enableServletCaching true}"
   ```

   Using Jython:

   ```
   serEnable = [['enableServletCaching', 'true']]
   ```

4. Enable caching, for example:

   Using Jacl:

   ```
   $AdminConfig modify $wc $serEnable
   ```

   Using Jython:

   ```
   AdminConfig.modify(wc, serEnable)
   ```

Configuring connections to Webservers with scripting

This topic contains the following tasks:

- "Regenerating the node plug-in configuration using scripting"
- "Creating new virtual hosts using templates with scripting" on page 156

Regenerating the node plug-in configuration using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

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 Jacl:

   ```
   set generator [$AdminControl completeObjectName type=PluginCfgGenerator,node=mynode,*]
   ```

   Using Jython:

   ```
   generator = AdminControl.completeObjectName('type=PluginCfgGenerator,node=mynode,*')
   ```

2. Regenerate the node plug-in:

   Using Jacl:

   ```
   $AdminControl invoke $generator generate "c:/WebSphere/DeloymentManager/config
   mycell mynode null plugin-cfg.xml"
   ```

   Using Jython:

```
AdminControl.invoke(generator, 'generate', "c:/WebSphere/DeloymentManager/config
mycell mynode null plugin-cfg.xml")
```

Creating new virtual hosts using templates with scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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*:
      `<WAS-ROOT>\config\templates\default\virtualhosts.xml`
   b. Edit and customize the new *myvirtualhostname.xml* file.
   c. Place the new file in the following directory:
      `<WAS-ROOT>\config\templates\custom\`

   If you want the new custom template to appear with the list of templates, restart the deployment manager on a network deployment edition, or use the AdminConfig object **reset** command. For example:
   - Using Jacl:
     ```
     $AdminConfig reset
     ```
   - Using Jython:
     ```
     AdminConfig.reset()
     ```

   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 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Managing servers with scripting

This topic contains the following tasks:

- “Stopping a node using scripting”
- “Starting servers using scripting”
- “Stopping servers using scripting” on page 158
- “Querying server state using scripting” on page 159
- “Listing running applications on running servers using scripting” on page 160
- “Starting listener ports using scripting” on page 162
- “Managing generic servers using scripting” on page 162
- “Setting development mode for server objects using scripting” on page 163
- “Disabling parallel startup using scripting” on page 164
- “Removing multicast endpoints using scripting” on page 164
- “Obtaining server version information with scripting” on page 165

Stopping a node using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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')
   ```

Starting servers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 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 milliseconds 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 milliseconds that the process should wait before starting the server.

Stopping servers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to query the server state:

1. 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.completObjectName('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
   ```

2. Query for the 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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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"
```

- 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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

A generic server is a server that the WebSphere Application Server manages but did not supply. You can use WebSphere Application Server to define, start, stop, and monitor generic servers.

- To define a generic server, use the following example:
  - Using Jacl:

```
$AdminTask createGenericServer mynode {-name generic1 -ConfigProcDef
{{"/usr/bin/myStartCommand" "arg1 arg2" "" "" "/tmp/workingDirectory"
"/tmp/stopCommand" "argy argz"}}}
$AdminConfig save
```

– Using Jython:

```
AdminTask.createGenericServer('mynode', '[-name generic1 -ConfigProcDef
[[c:\tmp\myStartCommand.exe "a b c" "" "" C:\tmp\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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Disabling parallel startup using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Removing multicast endpoints using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

WebSphere Application Server uses multicast broadcasting at the node level to allow a node agent to discover the managed processes in the node. The IPv4 and IPv6 multicast addresses are not compatible. Both the IPv4 and IPv6 multicast addresses are defined in the node agent configuration and when a node agent starts both addresses will be tried in sequence. It is recommended that you disable one of the multicast addresses after installation. By limiting multicast discovery to one protocol, the node agent will run more efficiently. Perform the following steps to remove a multicast endpoint:

1. Remove the multicast end point:
   - Using Jacl:
     ```
     set se [$AdminConfig getid /Node:y2001/ServerIndex:/]
     set eprs [lindex [$AdminConfig showAttribute $se endPointRefs] 0]
     foreach ep $eprs {
        set epName [$AdminConfig showAttribute $ep endPointName]
        if {$epName == "NODE_MULTICAST_DISCOVERY_ADDRESS"} {
     ```

```
        puts "Removing NODE_MULTICAST_DISCOVERY_ADDRESS..."
        $AdminConfig remove $ep
     }
  }
```

- Using Jython:

```
se = AdminConfig.getid('/Node:y2001/ServerIndex:/')
import java
lineseparator = java.lang.System.getProperty('line.separator')
eprs = AdminConfig.showAttribute(se, ['endPointRefs']).split(lineseparator)[0]
print eprs
for ep in eprs:
   epName = AdminConfig.showAttribute(ep, ['endPointName'])
   if (epName) == "[NODE_MULTICAST_DISCOVERY_ADDRESS]":
      print "Removing NODE_MULTICAST_DISCOVERY_ADDRESS..."
      AdminConfig.remove(ep)
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Obtaining server version information with scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 $server1 serverVersion
     ```

   - Using Jython:

     ```
     print AdminControl.getAttribute(server1, '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
---------------------------------------------------------------------
```

Clustering servers with scripting

This topic contains the following tasks:

- "Creating clusters using scripting"
- "Creating cluster members using scripting" on page 167
- "Starting a cluster using scripting" on page 168
- "Querying cluster state using scripting" on page 168
- "Stopping clusters using scripting" on page 169

Creating clusters using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

Perform the following steps to create a cluster:

1. Identify the server to convert to a cluster 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/')
   ```

2. Convert the existing server to a cluster by using the **convertToCluster** command passing in the existing server and the cluster name:

   Using Jacl:

   ```
   $AdminConfig convertToCluster $server myCluster1
   ```

   This command converts a cluster named myCluster with server1 as its member.

   Using Jython:

   ```
   print AdminConfig.convertToCluster(server, 'myCluster1')
   ```

   Example output:

   ```
   myCluster1(cells/mycell/cluster/myCluster1|cluster.xml#ClusterMember_1)
   ```

3. Save the configuration changes. See the "Saving configuration changes with the wsadmin tool" on page 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the "Synchronizing nodes with the wsadmin tool" on page 69 article for more information.

Creating cluster members using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

You can also use the AdminTask object to perform this task. For more about using the AdminTask object to create cluster members, see the Commands for AdminTask object article. To create cluster members using the AdminConfig object, perform the following steps:

1. 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)
   ```
2. 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)
   ```
3. (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)
   ```
4. 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)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Starting a cluster using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to start a cluster:

1. 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
   ```
2. 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.
3. 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
   ```
4. Start the cluster.
   - Using Jacl:
     ```
     $AdminControl invoke $cluster start
     ```
   - Using Jython:
     ```
     AdminControl.invoke(cluster, 'start')
     ```

   This command invokes the start operation on the Cluster MBean.

Querying cluster state using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to stop a cluster:

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.

Configuring security with scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

If you enable security for a WebSphere Application Server cell, supply authentication information to communicate with servers.

The sas.client.props and the soap.client.props files are located in the properties directory for each WebSphere Application Server profile, *profilePath*/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, or a Simple Object Access Protocol (SOAP) connector:
  - If you use a Remote Method Invocation (RMI) 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, a dialog box appears with a password prompt. If the script is running unattended, it appears to hang.
  - If you use a Simple Object Access Protocol (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=
    ```

- To 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. For example:

    ```
    wsadmin.sh -conntype RMI -port 2809 -user u1 -password secret1
    ```
  - Specify user name and password in the sas.client.props file for a RMI 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.

  Warning: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere profile.

Enabling and disabling global security using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

The default profile sets up procedures so that you can enable and disable global security based on LocalOS registry.

- You can use the **help** command to find out the arguments that you need to provide with this call, for example:
  - Using Jacl:

    ```
    securityon help
    ```

    Example output:

    ```
    Syntax: securityon user password
    ```
  - Using Jython:

    ```
    securityon()
    ```

    Example output:

    ```
    Syntax: securityon(user, password)
    ```

- To enable global security based on the LocalOS registry, use the following procedure call and arguments:
  - Using Jacl:

    ```
    securityon user1 password1
    ```
  - Using Jython:

    ```
    securityon('user1', 'password1')
    ```
- To disable global security based on the LocalOS registry, use the following procedure call:
  - Using Jacl:

    ```
    securityoff
    ```
  - Using Jython:

    ```
    securityoff()
    ```

Enabling and disabling Java 2 security using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

Perform the following steps to enable or disable Java 2 security:

1. Identify the security configuration object and assign it to the security variable:
   - Using Jacl:

     ```
     set security [$AdminConfig list Security]
     ```
   - Using Jython:

     ```
     security = AdminConfig.list('Security')
     print security
     ```

   Example output:

   ```
   (cells/mycell|security.xml#Security_1)
   ```
2. Modify the enforceJava2Security attribute to enable or disable Java 2 security. For example:
   - 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']])
       ```
3. Save the configuration changes. See the "Saving configuration changes with the wsadmin tool" on page 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the "Synchronizing nodes with the wsadmin tool" on page 69 article for more information.

Configuring data access with scripting

This topic contains the following tasks:

Configuring a JDBC provider using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new JDBC provider:

1. 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)
   ```
2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required JDBCProvider
     ```
   - Using Jython:

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

   Example output:

   ```
   Attribute     Type
   name     String
   implementationClassName   String
   ```
3. 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']]
```

4. 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)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new data sources using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new data source:

1. Identify the parent ID:
   - 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:
   - Using Jacl:

     ```
     $AdminConfig required DataSource
     ```

   - Using Jython:

     ```
     print AdminConfig.required('DataSource')
     ```

   Example output:

   ```
   Attribute   Type
   name    String
   ```

3. Setting up required attributes:
   - Using Jacl:

     ```
     set name [list name DS1]
     set dsAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'DS1']
     dsAttrs = [name]
     ```

4. Create a 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)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new connection pools using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new connection pool:

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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new data source custom properties using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new data source custom property:g

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2CAuthentication data entries using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new J2CAuthentication 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WAS40 data sources using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new WAS40 data source:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjdbc [$AdminConfig getid "/JDBCProvider:Cloudscape JDBC Provider/"]
     ```

   - Using Jython:

     ```
     newjdbc = AdminConfig.getid('/JDBCProvider:Cloudscape 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WAS40 connection pools using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WAS40 custom properties using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2C resource adapters using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new J2C resource adapter:

1. 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)
   ```
2. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required J2CResourceAdapter
     ```
   - Using Jython:
     ```
     print AdminConfig.required('J2CResourceAdapter')
     ```

   Example output:
   ```
   Attribute     Type
   name         String
   ```
3. Set up the required attributes:
   - Using Jacl:
     ```
     set rarFile c:/currentScript/cicseci.rar
     set option  [list  -rar.name  RAR1]
     ```
   - Using Jython:
     ```
     rarFile = 'c:/currentScript/cicseci.rar'
     option  = '[-rar.name  RAR1]'
     ```
4. Create a resource adapter:
   - Using Jacl:
     ```
     set newra [$AdminConfig installResourceAdapter $rarFile mynode $option]
     ```
   - Using Jython:
     ```
     newra = AdminConfig.installResourceAdapter(rarFile, 'mynode', option)
     print newra
     ```

   Example output:
   ```
   RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring custom properties for J2C resource adapters using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2C connection factories using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 Jacl:

          ```
          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 Jacl:

          ```
          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 j2ccfAttrs [list $name]
          set jname [list jndiName eis/j2ccf1]
          ```

        - Using Jython:

          ```
          name = ['name', 'J2CCF1']
          j2ccfAttrs = [name]
          jname = ['jndiName', eis/j2ccf1]
          ```

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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2C authentication data entries using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new J2C authentication data entry:

1. Identify the parent ID and assign it to the security variable.
   - Using Jacl:
     ```
     set security [$AdminConfig getid /Security:mysecurity/]
     ```
   - Using Jython:
     ```
     security = AdminConfig.getid('/Security:mysecurity/')
     ```
2. Identify the 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 the 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]
```

Example output:

```
[[alias, myAlias], [userId, myid], [password, secret]]
```

4. Create JAAS authentication data:
   - Using Jacl:

     ```
     $AdminConfig create JAASAuthData $security $jaasAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('JAASAuthData', security, jaasAttrs)
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#JAASAuthData_2)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2C activation specs using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a J2C activation specs:

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 -message ListenerType
        javax.jms.MessageListener}
        ```

        Using Jython:

        ```
        AdminTask.createJ2CActivationSpec(newra, ['-name', 'ao1',
        '-jndiName', 'eis/ao1', '-message', 'ListenerType',
        '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 specs defined, it is required that you specify the activation spec attribute. If your resource adapter is JCA V1.5 and you have only one activation spec 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 specs 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 ActivationSpecID
     set name [list name J2CAC1]
     set jname [jndiName eis/j2cac1]
     set j2cacAttrs [list $name $jname]
     ```

     Using Jython:

     ```
     ac = ActivationSpecID
     name = ['name', 'J2CAC1']
     jname = ['jndiName', 'eis/j2cac1']
     j2cacAttrs = [name, jname]
     ```

  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 spec 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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new J2C administrative objects using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Testing data source connections using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to test a data source to ensure a connection to the database.

1. 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.
   ```

2. 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:

     ```
     $AdminControl invoke $dshelper testConnectionToDataSource
     "COM.ibm.db2.jdbc.DB2XADataSource db2admin db2admin
     {{databaseName sample}} c:/sqllib/java/db2java.zip en US"
     ```

   - Using Jython:

     ```
     print AdminControl.invoke(dshelper, 'testConnectionToDataSource',
     'COM.ibm.db2.jdbc.DB2XADataSource dbuser1 dbpwd1
     "{{databaseName jtest1}}" c:/sqllib/java12/db \"\" \"\"')
     ```

   Example output:

   ```
   WASX7217I: Connection to provided data source was successful.
   ```

Configuring messaging with scripting

This topic contains the following tasks:

Configuring the message listener service using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 103 article for more information.

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 84 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new JMS providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new JMS destinations using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new JMS connections using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new JMS connection:

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 GenericJMSConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('GenericJMSConnectionFactory')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   externalJNDIName  String
   ```

3. 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]]
```

4. 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)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WebSphere queue connection factories using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new WebSphere queue connection factory:

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 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. Create was queue connection factories:
   - Using Jacl:

     ```
     $AdminConfig create WASQueueConnectionFactory $newjmsp $mqcfAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASQueueConnectionFactory', newjmsp, mqcfAttrs)
     ```

   Example output:

   ```
   WASQCF(cells/mycell/nodes/mynode|resources.xml#WASQueueConnectionFactory_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WebSphere topic connection factories using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new WebSphere topic connection factory:

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 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 $newjmsp $mtcfAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASTopicConnectionFactory', newjmsp, 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WebSphere queues using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new WebSphere queue:

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 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 $newjmsp $wqAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASQueue', newjmsp, 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new WebSphere topics using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new WebSphere topic:

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 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 $newjmsp $wtAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASTopic', newjmsp, 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new MQ queue connection factories using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new MQ queue connection factory:

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 MQQueueConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQQueueConnectionFactory')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```

3. Set up 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 MQ queue connection factory:
   - 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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new MQ topic connection factories using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new MQ topic connection factory:

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 MQTopicConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQTopicConnectionFactory')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```

3. Set up 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 mq topic connection factory:
   - Using Jacl:

     ```
     $AdminConfig create MQTopicConnectionFactory $newjmsp $mqtcfAttrs $template
     ```

   - Using Jython:

     ```
     print AdminConfig.create('MQTopicConnectionFactory', newjmsp, 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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new MQ queues using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new MQ queue:

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 MQQueue
     ```
   - Using Jython:

     ```
     print AdminConfig.required('MQQueue')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   baseQueueName  String
   ```
3. Set up 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 MQ queue factory:
   - Using Jacl:
     ```
     $AdminConfig create MQQueue $newjmsp $mqqAttrs $template
     ```
   - Using Jython:
     ```
     print AdminConfig.create('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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new MQ topics using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to configure a new MQ topic:

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 MQTopic
     ```
   - Using Jython:
     ```
     print AdminConfig.required('MQTopic')
     ```

   Example output:
   ```
   Attribute            Type
   name                 String
   jndiName             String
   baseTopicName        String
   ```
3. Set up 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 MQ topic factory:
   - 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring mail, URLs, and resource environment entries with scripting

This topic contains the following tasks:

Configuring new mail providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new mail sessions using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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.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 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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new protocols using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new custom properties using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new resource environment providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring custom properties for resource environment providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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)
   ```

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. See the “Saving configuration changes with the wsadmin tool” on page 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new referenceables using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new resource environment entries using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring custom properties for resource environment entries using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new URL providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring custom properties for URL providers using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring new URLs using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Configuring custom properties for URLs using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Troubleshooting with scripting

This topic contains the following tasks:

Tracing operations with the wsadmin tool

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to trace operations:

Enable wsadmin client tracing with the following command:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=all=enabled
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=all=enabled')
  ```

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=enabled`	indicates to turn on tracing

The following command disables tracing:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=all=disabled
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=all=disabled')
  ```

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=disabled`	indicates to turn off tracing

The trace command changes the trace settings for the current session. You can change this setting persistently by editting 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.

Configuring traces using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

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 84 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 69 article for more information.

Turning traces on and off in servers processes using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Perform the following steps to turning 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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 article for more information.

Use the AdminControl object to dump the Java threads of a running server. 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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 103 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 103 article.

Scripting reference material

This topic contains the following tasks:

Wsadmin tool

The WebSphere Application Server wsadmin tool runs scripts. You can use the wsadmin tool to manage WebSphere Application Server as well as the configuration, application deployment, and server run-time operations.

The command-line invocation syntax for the wsadmin scripting client is as follows:

```
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 userid] [-password password] |

     RMI [-host host_name] [-port port_number] [-user userid] [-password password] |

     NONE
]

[script parameters]
```

Where *script parameters* represent any arguments other than the ones listed previously. The argc variable contains the argument number, and the argv variable contains the contents.

Options

-c Designates to run a single command.

Multiple -c options can exist on the command line. They run in the order that you designate. You must save after using this command.

-f Designates a script to run.

Only one -f option can exist on the command line.

-javaoption
Specifies a valid Java standard or a non-standard option.

Multiple -javaoption options can exist on the command line.

-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, or the com.ibm.ws.scripting.defaultLang property. The `-lang` argument has no default value. If the command line or the property does not supply the script language, and the wsadmin tool cannot determine it, an error message generates. This argument is required if not determined from the script file name.

-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 WebSphere Application Server `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 that the environment variable `WSADMIN_PROPERTIES` points to.

Multiple -p options can exist on the command line. They 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`, the profile script runs before it invokes this command. If you specify `-f`, 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 will run. 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.

-?
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`, and `NONE`.

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.

-wsadmin_classpath
Use this option to make additional classes available to your scripting process.

Follow this option with a class path string. For example:

```
c:/MyDir/Myjar.jar;d:/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 hostname to which wsadmin should attempt to connect. The default wsadmin.properties file located in the properties directory of each WebSphere 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.

Warning: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere profile.

-username
Specify a user name to be used by the connector to connect to the server if security is enabled in the server.

-port
Specify a port to be used by the connector. The default wsadmin.properties file located in the properties directory of each WebSphere Application Server profile provides a value in the port property to connect to the local server.

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:

```
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 -javaoption -Xmx256m -f test.jacl
```

Using Jython:

```
wsadmin -lang jython -javaoption -Xms128m -javaoption -Xmx256m -f test.py
```

RMI connection with security

Using Jacl:

```
wsadmin -conntype RMI -port  rmiportnumber -userid userid
-password password
```

Using Jython:

```
wsadmin -lang jython -conntype RMI -port  rmiportnumber -userid userid
-password password
```

Warning: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere profile.

Local mode of operation to perform a single command

Using Jacl:

```
wsadmin -conntype NONE -c "$AdminApp uninstall app"
```

or

Using Jython:

```
wsadmin -lang jython -conntype NONE -c "AdminApp.uninstall('app')"
```

or

wsadmin tool performance tips

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.bat` or the `wsadmin.sh` files, 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 "$AdminApp install c:\\myApps\\App1.ear {-appname appl1}"
  wsadmin -c "$AdminApp install c:\\myApps\\App2.ear {-appname appl2}"
  wsadmin -c "$AdminApp install c:\\myApps\\App3.ear {-appname appl3}"
  ```

  or

  Using Jython:

  ```
  wsadmin -lang jython -c "AdminApp.install('c:\myApps\App1.ear', '[-appname appl1]')"
  wsadmin -lang jython -c "AdminApp.install('c:\myApps\App2.ear', '[-appname appl2]')"
  wsadmin -lang jython -c "AdminApp.install('c:\myApps\App3.ear', '[-appname appl3]')"
  ```

  or

  Or, for example, using Jacl, you can create the *appinst.jacl* file that contains the commands:

  ```
  $AdminApp install c:\\myApps\\App1.ear {-appname appl1}
  $AdminApp install c:\\myApps\\App2.ear {-appname appl2}
  $AdminApp install c:\\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('c:\myApps\App1.ear', '[-appname appl1]')
AdminApp.install('c:\myApps\App2.ear', '[-appname appl2]')
AdminApp.install('c:\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

The Help object provides general help and dynamic online information about the currently running MBeans. You can 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:

Command name:	Description:	Parameters and return values:	Examples:

AdminApp	Provides a summary of all of the available methods for the AdminApp object.	• Parameters: none • Returns: string	Example usage: Using Jacl: `$Help AdminApp` Using Jython: `print Help.AdminApp()` Example 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. 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. 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

Admin Config	Provides a summary of all the available methods for the Admin Config object.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$Help AdminConfig` Using Jython: `print Help.AdminConfig()` Example 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`

Admin Control	Provides a summary of the help commands and ways to invoke an administrative command.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$Help AdminControl` Using Jython: `print Help.AdminControl()` Example 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

AdminTask	Provides a summary of help commands and ways to invoke an administrative command.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$AdminTask help` Using Jython: `print AdminTask.help()` Example output: `WASX8001I: The AdminTask object enables the available administrative commands. AdminTask commands operate in two modes: the default mode is one which AdminTask 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.

all	Provides a summary of the information that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example 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 setTraceOutputTo RingBuffer`

			(int, java.lang.String) java.lang.String rolloverLogFileImmediate (java.lang.String, java.lang.String) Notifications jmx.attribute.changed Constructors
attributes	Provides a summary of all the attributes that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Attribute Type Access` `ringBufferSize java. lang.Integer RW` `traceSpecification string RW`
classname	Provides a class name that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `javax.management.modelmbean. RequiredModelMBean`

constructors	Provides a summary of all of the constructors that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Constructors`
description	Provides a description that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Managed object for overall server process.`

help	Provides a summary of all the available methods for the Help object.	• Parameters: None • Returns: string	Example 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
message	Displays information for a message ID.	• Parameters: message ID • Returns: string	Example usage: Using Jacl: `$Help message CNTR0005W` Using Jython: `print Help.message('CNTR0005W')` Example output: `Explanation: The container was unable to passivate an enterprise bean due to exception {2}` `User action: Take action based upon message in exception {2}`
notifications	Provides a summary of all the notifications that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: Using Jacl: `$Help notifications [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]` Using Jython: `print Help.notifications (AdminControl.queryNames ('type=TraceService,process=server1,node=pongo,*'))` Example output: `Notification` `websphere.messageEvent.audit` `websphere.messageEvent.fatal` `websphere.messageEvent.error` `websphere.seriousEvent.info` `websphere.messageEvent.warning` `jmx.attribute.changed`

operations	Provides a summary of all the operations that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: Using Jacl: `$Help operations [$AdminControl queryNames type=TraceService,process= server1,node=pongo,*]` Using Jython: `print Help.operations (AdminControl.queryNames ('type=TraceService,process= server1,node=pongo,*'))` Example output: `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 getTrace Specification(java.lang.String) void processDumpString (java.lang.String) void checkTraceString (java.lang.String) void setTraceOutputToFile (java.lang.String, int, int, java.lang.String) void setTraceOutputToRin gBuffer(int, java.lang.String) java.lang.String rolloverLogFileImmediate (java.lang.String, java.lang.String)`

operations	Provides the signature of the opname operation for the MBean that is defined by name.	• Parameters: name - string, opname - string • Returns: string	Example usage: Using Jacl: `$Help operations [$AdminControl queryNames type=TraceService,process= server1,node=pongo,*] processDumpString` Using Jython: `print Help.operations (AdminControl.queryNames ('type=TraceService,process= server1,node=pongo,*'), 'processDumpString')` Example output: `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.`

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:

Command name:	Description:	Parameters and return values:	Examples:

attributes	Returns a list of the top level attributes for a given type.	• Parameters: object type The name of the object type that you input here is the one based on the XML configuration files and does not have to be the same name that the administrative console displays. • Returns: A list of attributes.	Example usage: Using Jacl: `$AdminConfig attributes ApplicationServer` Using Jython: `print AdminConfig.attributes ('ApplicationServer')` Example output: `"properties Property*" "serverSecurity ServerSecurity" "server Server@" "id Long" "stateManagement StateManageable" "name String" "moduleVisibility EEnumLiteral(MODULE, COMPATIBILITY, SERVER, APPLICATION)" "services Service*" "statisticsProvider StatisticsProvider"`
checkin	Checks a file that the document URI describes into the configuration repository. This method only applies to deployment manager configurations.	• Parameters: document URI, filename, opaque object • Returns: None	Example usage: Using Jacl: `$AdminConfig checkin cells /MyCell/Node/MyNode/ serverindex.xml c:\\mydir\myfile $obj` Using Jython: `AdminConfig.checkin('cells/ MyCell/Node/MyNode/ serverindex.xml', 'c:\mydir\myfile', obj)` The document URI is relative to the root of the configuration repository, for example, `c:\WebSphere\AppServer\config`. The file that is specified by `filename` is used as the source of the file to check. The *opaque* object is an object that the **extract** command of the AdminConfig object returns by a prior call.

convert ToCluster	Converts a server so that it is the first member of a new server cluster.	• Parameters: server ID, cluster name • Returns: The configuration ID of the new cluster.	Example usage: Using Jacl: `set serverid [$AdminConfig getid /Server:myServer/] $AdminConfig convertToCluster $serverid myCluster` Using Jython: `serverid = AdminConfig. getid('/Server:myServer/') AdminConfig.convertToCluster (serverid, 'myCluster')` Example output: `myCluster(cells/mycell/ clusters/myCluster|cluster. xml#ClusterMember_2`
create	Creates configuration objects.	• Parameters using Jacl: type- string; parent ID- string; attributes- string • Parameters using Jython: type- string; parent ID- string; attributes- string or type- string; parent ID- string; attributes- Jython list • Returns: A string with configuration object names.	The name of the object type that you input here is the one that is based on the XML configuration files. This name does not have to be the same name that the administrative console displays. Example usage: Using Jacl: `set jdbc1 [$AdminConfig getid /JDBCProvider:jdbc1/] $AdminConfig create DataSource $jdbc1 {{name ds1}}` Using Jython with string attributes: `jdbc1 = AdminConfig.getid ('/JDBCProvider:jdbc1/') AdminConfig.create ('DataSource', jdbc1, '[[name ds1]]')` Using Jython with object attributes: `jdbc1 = AdminConfig.getid ('/JDBCProvider:jdbc1/') AdminConfig.create ('DataSource', jdbc1, [['name', 'ds1']])` Example output: `ds1(cells/mycell/nodes/ DefaultNode/servers/ server1|resources.xml# DataSource_6)`

createCluster Member	Creates a new server as a member of an existing cluster. This method creates 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.	• Parameters using Jacl: cluster ID- string; node ID- string; member attributes- string • Parameters using Jython: cluster ID- string; node ID- string; member attributes- string or cluster ID- string; node ID- string; member attributes- Jython list • Returns: The configuration ID of the new cluster member.	The name of the object type that you input here is the one that is based on the XML configuration files. This name does not have to be the same name that the administrative console displays. Example usage: Using Jacl: `set clid [$AdminConfig getid /ServerCluster: myCluster/] set nodeid [$AdminConfig getid /Node:mynode/] $AdminConfig create ClusterMember $clid $nodeid {{memberName newMem1} {weight 5}}` Using Jython with string attributes: `clid = AdminConfig.getid ('/ServerCluster: myCluster/') nodeid = AdminConfig. getid('/Node:mynode/') AdminConfig.create ClusterMember(clid, nodeid, '[[memberName newMem1] [weight 5]]')` Using Jython with object attributes: `clid = AdminConfig.getid ('/ServerCluster: myCluster/') nodeid = AdminConfig. getid('/Node:mynode/') AdminConfig.createCluster Member(clid, nodeid, [['memberName', 'newMem1'], ['weight', 5]])` Example output: `myCluster(cells/mycell/ clusters/myCluster| cluster.xml#ClusterMember_2)`

create Document	Creates a new document in the configuration repository. The documentURI parameter names the document to create in the repository. The filename parameter must be a valid local file name where the contents of the document exist.	• Parameters: documentURI, filename • Returns: None	Example usage: Using Jacl: `$AdminConfig createDocument cells/mycell/myfile.xml c:\\mydir\\myfile` Using Jython: `AdminConfig.createDocument ('cells/mycell/myfile.xml', 'c:\mydir\myfile')`
`createUsing Template`	Creates a type of object with the given parent, using a template.	• Parameters using Jacl: type-string; parent id-string; attributes-string; template ID-string • Parameters using Jython: type-string; parent id- string; attributes-string; template ID-string or type-string; parent id-string; attributes-Jython list; template ID-string • Returns: The configuration ID of a new object.	Example usage: Using Jacl: `set node [$AdminConfig getid /Node:`*`mynode/`*`] set templ [$AdminConfig listTemplates JDBCProvider "DB2 JDBC Provider (XA)"] $AdminConfig createUsing Template JDBCProvider $node {{name `*`newdriver`*`}} $templ` Using Jython using string attributes: `node = AdminConfig.getid ('/Node:mynode/') templ = AdminConfig. listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)") AdminConfig.createUsing Template('JDBCProvider', node, '[[name newdriver]]', templ)` Using Jython using object attributes: `node = AdminConfig.getid ('/Node:mynode/') templ = AdminConfig. listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)") AdminConfig.createUsing Template('JDBCProvider', node, [['name', 'newdriver']], templ)`

defaults	Displays 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.	• Parameters: type The name of the object type that you input here is the one based on the XML configuration files. This name does not have to be the same name that the administrative console displays. • Returns: A string that contains a list of attributes with its type and value.	Example usage: Using Jacl: `$AdminConfig defaults TuningParams` Using Jython: `print AdminConfig.defaults ('TuningParams')` Example output: `Attribute Type Default` `usingMultiRowSchema Boolean false maxInMemorySessionCount Integer 1000 allowOverflow Boolean true scheduleInvalidation Boolean false writeFrequency ENUM writeInterval Integer 120 writeContents ENUM invalidationTimeout Integer 30 invalidationSchedule InvalidationSchedule`
delete Document	Deletes a document from the configuration repository. The documentURI parameter names the document to delete from the repository.	• Parameters: documentURI • Returns: None	Example usage: Using Jacl: `$AdminConfig deleteDocument cells/mycell/myfile.xml` Using Jython: `AdminConfig.delete Document('cells/mycell /myfile.xml')`
exists Document	Tests for the existence of a document in the configuration repository. The documentURI parameter names the document to test in the repository.	• Parameters: documentURI • Returns: A true value, if the document exists.	Example usage: Using Jacl: `$AdminConfig existsDocument cells/mycell/myfile.xml` Using Jython: `AdminConfig.existsDocument ('cells/mycell/myfile.xml')` Example output: `1`

extract	Extracts 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.	• Parameters: document URI, filename • Returns: An opaue java.lang.Object to use when checking in the file.	Example usage: Using Jacl: `set obj [$AdminConfig extract cells/MyCell/ nodes/MyNode/serverindex. xml c:\\mydir\myfile]` Using Jython: `obj = AdminConfig.extract ('cells/MyCell/nodes/ MyNode/serverindex.xml', 'c:\mydir\myfile')` The document URI is relative to the root of the configuration repository, for example, `c:\WebSphere\AppServer\config`. If the file that is specified by the filename parameter exists, the extracted file replaces it.
getCross Document Validation Enabled	Returns a message with the current cross-document enablement setting. This method returns true if cross-document validation is enabled.	• Parameters: None • Returns: A string that contains the message with the cross-document validation setting.	Example usage: Using Jacl: `$AdminConfig getCross DocumentValidationEnabled` Using Jython: `print AdminConfig.getCross DocumentValidationEnabled()` Example output: `WASX7188I: Cross-document validation enablement set to true`
getid	Returns the configuration ID of an object.	• Parameters: containment path • Returns: The configuration ID for an object that is described by the containment path.	Example usage: Using Jacl: `$AdminConfig getid /Cell: testcell/Node:testNode/ JDBCProvider:Db2JdbcDriver/` Using Jython: `AdminConfig.getid('/Cell: testcell/Node:testNode/ JDBCProvider:Db2JdbcDriver/')` Example output: `Db2JdbcDriver(cells/ testcell/nodes/testnode| resources.xml# JDBCProvider_1)`

getObjectName	Returns a string version of the object name for the corresponding running MBean. This method returns an empty string if no corresponding running MBean exists.	• Parameters: configuration ID • Returns: A string that contains the object name.	Example usage: Using Jacl: `set server [$AdminConfig getid /Node:mynode/ Server:server1/] $AdminConfig getObjectName $server` Using Jython: `server = AdminConfig. getid('/Node:mynode/ Server:server1/') AdminConfig.getObject Name(server)` Example output: `WebSphere:cell=mycell, name=server1,mbean Identifier=cells/ mycell/nodes/mynode/ servers/server1/server. xml#Server_1, type=Server,node=mynode, process=server1,process Type=UnManagedProcess`
getSaveMode	Returns the mode that is used when you invoke a **save** command. Possible values include the following: • overwrite OnConflict - Saves changes even if they conflict with other configuration changes • rollback OnConflict - Causes a save operation to fail if changes conflict with other configuration changes. This value is the default.	• Parameters: None • Returns: A string that contains the current save mode setting.	Example usage: Using Jacl: `$AdminConfig getSaveMode` Using Jython: `print AdminConfig. getSaveMode()` Example output: `rollbackOnConflict`

getValidation Level	Returns the validation used when files are extracted from the repository.	• Parameters: None • Returns: A string that contains the validation level.	Example usage: Using Jacl: `$AdminConfig getValidationLevel` Using Jython: `AdminConfig.getValidationLevel()` Example output: `WASX7189I: Validation level set to HIGH`
getValidation SeverityResult	Returns the number of validation messages with the given severity from the most recent validation.	• Parameters: severity • Returns: A string that indicates the number of validation messages of the given severity.	Example usage: Using Jacl: `$AdminConfig getValidation SeverityResult 1` Using Jython: `AdminConfig.getValidation SeverityResult(1)` Example output: `16`
hasChanges	Returns `true` if unsaved configuration changes exist.	• Parameters: None • Returns: A string that indicates whether unsaved configuration changes exist.	Example usage: Using Jacl: `$AdminConfig hasChanges` Using Jython: `AdminConfig.hasChanges()` Example output: `1`

help	Displays static help information for the AdminConfig object.	• Parameters: None • Returns: A list of options.	Example usage: Using Jacl: `$AdminConfig help` Using Jython: `print AdminConfig.help()` Example output: `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 setCrossDocument ValidationEnabled 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
install Resource Adapter	Installs a Java 2 Connector (J2C) resource adapter with the given Resource Adapter Archive (RAR) file name and an option string in the node. The RAR file name is the fully qualified file name that resides in the node that you specify. The valid options include the following options: • rar.name • rar.desc • rar.archivePath • rar.classpath • rar.nativePath • rar.threadPoolAlias • rar.propertiesSet	• Parameters: RAR file name, node, options • Returns: The configuration ID of the new J2CResourceAdapter object.	Example usage: Using Jacl: `$AdminConfig installResource Adapter c:/rar/mine.rar mynode {-rar.name myResourceAdapter -rar.desc "My rar file"}` Using Jython: `print AdminConfig.install ResourceAdapter('c:/rar/mine.rar', 'mynode', '[-rar.name myResource Adapter -rar.desc "My rar file"]')` Example output: `myResourceAdapter(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)`

	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 {}.		

	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.		
list	Returns a list of objects of a given type, possibly scoped by a parent.	• Parameters: 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 the administrative console displays. • Returns: A list of objects.	Example usage: Using Jacl: `$AdminConfig list JDBCProvider` Using Jython: `print AdminConfig.list ('JDBCProvider')` Example output: `Db2JdbcDriver(cells/mycell/ nodes/DefaultNode|resources. xml#JDBCProvider_1) Db2JdbcDriver(cells/mycell/ nodes/DefaultNode/servers/ deploymentmgr|resources. xml#JDBCProvider_1) Db2JdbcDriver(cells/mycell/ nodes/DefaultNode/servers/ nodeAgent|resources.xml# JDBCProvider_1)`
listTemplates	Displays a list of template object IDs.	• Parameters: object type The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that the administrative console displays. • Returns: A list of template IDs.	Example usage: Using Jacl: `$AdminConfig listTemplates JDBCProvider` Using Jython: `print AdminConfig.list Templates('JDBCProvider')` This example displays a list of all the JDBCProvider templates that are available on the system.

modify	Supports the modification of object attributes.	• Parameters using Jacl: object-string; attributes-string • Parameters using Jython: object-string; attributes-string or object-string; attributes-Jython list • Returns: None	Example usage: Using Jacl: `$AdminConfig modify ConnFactory1(cells/ mycell/nodes/Default Node/servers/ deploymentmgr|resources. xml#GenericJMSConnection Factory_1) {{userID newID} {password newPW}}` Using Jython with string attributes: `AdminConfig.modify ('ConnFactory1(cells/ mycell/nodes/DefaultNode/ servers/deploymentmgr| resources.xml#Generic JMSConnectionFactory_1)', '[[userID newID] [password newPW]]')` Using Jython with object attributes: `AdminConfig.modify ('ConnFactory1(cells/ mycell/nodes/DefaultNode/ servers/deploymentmgr| resources.xml#Generic JMSConnectionFactory_1)', [['userID', 'newID'], ['password', 'newPW']])`
parents	Obtains information about object types.	• Parameters: object type The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that the administrative console displays. • Returns: A list of object types.	Example usage: Using Jacl: `$AdminConfig parents JDBCProvider` Using Jython: `AdminConfig.parents ('JDBCProvider')` Example output: `Cell Node Server`
query Changes	Returns a list of unsaved configuration files.	• Parameters: None • Returns: A string that contains a list of files with unsaved changes.	Example usage: Using Jacl: `$AdminConfig queryChanges` Using Jython: `AdminConfig.queryChanges()` Example output: `WASX7146I: The following configuration files contain unsaved changes: cells/mycell/nodes/mynode/ servers/server1|resources.xml`

remove	Removes a configuration object.	• Parameters: Object • Returns: None	Example usage: Using Jacl: `$AdminConfig remove ds1(cells/mycell/nodes/ DefaultNode/servers/ server1:resources.xml# DataSource_6)` Using Jython: `AdminConfig.remove('ds1 (cells/mycell/nodes/ DefaultNode/servers/ server1:resources.xml# DataSource_6)')`
required	Displays the required attributes that are contained by an object of a certain type.	• Parameters: Type The name of the object type that you input here is the one that is based on the XML configuration files. It does not have to be the same name that the administrative console displays. • Returns: A string that contains a list of the required attributes with its type.	Example usage: Using Jacl: `$AdminConfig required URLProvider` Using Jython: `print AdminConfig.required ('URLProvider')` Example output: `Attribute Type streamHandlerClassName String protocol String`
reset	Resets the temporary workspace that holds updates to the configuration.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminConfig reset` Using Jython: `AdminConfig.reset()`
save	Saves changes in the configuration repository.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminConfig save` Using Jython: `AdminConfig.save()`
setCross Document Validation Enabled	Sets the cross-document validation enabled mode. Values include `true` or `false`.	• Parameters: Flag • Returns: None	Example usage: Using Jacl: `$AdminConfig setCrossDocument ValidationEnabled true` Using Jython: `AdminConfig.setCrossDocument ValidationEnabled('true')`

setSaveMode	Toggles the behavior of the **save** command. The default value is `rollbackOnConflict`. When a conflict is discovered 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.	• Parameters: Mode • Returns: None	Example usage: Using Jacl: `$AdminConfig setSaveMode overwriteOnConflict` Using Jython: `AdminConfig.setSaveMode ('overwriteOnConflict')`
setValidation Level	Sets the validation that is used when files are extracted from the repository. Five validation levels are available: `none`, `low`, `medium`, `high`, or `highest`.	• Parameters: Level • Returns: A string that contains the validation level setting.	Example usage: Using Jacl: `$AdminConfig setValidationLevel high` Using Jython: `AdminConfig.setValidationLevel ('high')` Example output: `WASX7189I: Validation level set to HIGH`

show	Returns the top-level attributes of the given object.	• Parameters: Object, attributes • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `$AdminConfig show Db2JdbcDriver (cells/mycell/nodes/ DefaultNode|resources.xm# JDBCProvider_1)` Example output with Jacl: `{name "Sample Datasource"} {description "Data source for the Sample entity beans"}` Using Jython: `print AdminConfig.show ('Db2JdbcDriver(cells/ mycell/nodes/DefaultNode| resources.xm#JDBCProvider_1)')` Example output with Jython: `[name "Sample Datasource"] [description "Data source for the Sample entity beans"]`

showall	Recursively shows the attributes of a given configuration object.	• Parameters: Object, attributes • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `$AdminConfig showall "Default Datasource (cells/mycell/nodes/ DefaultNode/servers/ server1:resources.xml# DataSource_1)` Example output with Jacl: `{authMechanismPreference BASIC_PASSWORD} {category default} {connectionPool {{agedTimeout 0} {connectionTimeout 1000} {maxConnections 30} {minConnections 1} {purgePolicy Failing ConnectionOnly} {reapTime 180} {unusedTimeout 1800}}} {datasourceHelperClassname com.ibm.websphere. rsadapter.CloudscapeData StoreHelper} {description "Datasource for the WebSphere Default Application"} {jndiName DefaultDatasource} {name "Default Datasource"} {propertySet {{resource Properties {{{description "Location of Cloudscape 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 "Cloudscape 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} Using Jython: AdminConfig.showall ("Default Datasource (cells/mycell/nodes/ DefaultNode/servers/ server1:resources.xml# DataSource_1)") Example output with Jython: [authMechanismPreference BASIC_PASSWORD] [category default] [connectionPool [[agedTimeout []] [connectionTimeout 1000] [maxConnections 30] [minConnections 1] [purgePolicy Failing ConnectionOnly] [reapTime 180] [unusedTimeout 1800]]] [datasourceHelperClassname com.ibm.websphere. rsadapter.CloudscapeData StoreHelper] [description "Datasource for the WebSphere Default Application"] [jndiName DefaultDatasource] [name "Default Datasource"] [propertySet [[resource Properties [[[description "Location of Cloudscape 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 "Cloudscape 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]
show Attribute	Displays only the value for the single attribute that you specify. 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.	• Parameters: Configuration ID, attribute • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `set ns [$AdminConfig getid /Node:mynode/] $AdminConfig showAttribute $ns hostName` Using Jython: `ns = AdminConfig.getid ('/Node:mynode/') print AdminConfig.show Attribute(ns, 'hostName')` Example output: `mynode`

types	Returns a list of the configuration object types that you can manipulate.	• Parameters: None • Returns: A list of object types.	Example usage: Using Jacl: `$AdminConfig types` Using Jython: `print AdminConfig.types()` Example output: `AdminService` `Agent` `ApplicationConfig` `ApplicationDeployment` `ApplicationServer` `AuthMechanism` `AuthenticationTarget` `AuthorizationConfig` `AuthorizationProvider` `AuthorizationTableImpl` `BackupCluster` `CMPConnectionFactory` `CORBAObjectNameSpaceBinding` `Cell` `CellManager` `Classloader` `ClusterMember` `ClusteredTarget` `CommonSecureInteropComponent`

uninstall Resource Adapter	Uninstalls a Java 2 Connector (J2C) resource adapter with the given J2C resource adapter configuration ID and an option list. One option is valid for this command: * `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.	• Parameters: J2C resource adapter configuration ID, list of options • Returns: The configuration ID of J2CResourceAdapter object that is removed.	Example usage: Using Jacl: `set j2cra [$AdminConfig getid /J2CResourceAdapter:MyJ2CRA/] $AdminConfig uninstallResource Adapter $j2cra {-force} $AdminConfig save` Using Jython: `j2cra = AdminConfig.getid ('/J2CResourceAdapter: MyJ2CRA/') print AdminConfig.uninstall ResourceAdapter(j2cra, '[-force]') AdminConfig.save()` Example output: `WASX7397I: The following J2CResourceAdapter objects are removed: MyJ2CRA(cells/juniarti/ nodes/juniarti|resources. xml#J2CResourceAdapter_ 1069433028609)`
	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.		

validate	Invokes validation. This command requests 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 config id parameter.	• Parameters: config id (optional) • Returns: A string that contains results of the validation.	Example usage: Using Jacl: `$AdminConfig validate` Using Jython: `print AdminConfig.validate()` Example output: `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`

Commands for the AdminControl object

Use the AdminControl object to invoke operational commands that deal with running objects in the WebSphere 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:

Command name:	Description:	Parameters and return values:	Examples:

complete ObjectName	Creates 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 it finds several MBeans that match the fragment, the command returns the first one.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set serverON [$AdminControl completeObjectName node= mynode,type=Server,*]` Using Jython: `serverON = AdminControl. completeObjectName('node= mynode,type=Server,*')`
getAttribute	Returns the value of the attribute for the name that you provide.	• Parameters: name-java.lang.String; attribute-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] $AdminControl getAttribute $objNameString processType` Using Jython: `objNameString = Admin Control.completeObject Name('WebSphere:type= Server,*') AdminControl.getAttribute (objNameString, 'processType')`

getAttribute _jmx	Returns the value of the attribute for the name that you provide.	• Parameters: name-ObjectName; attribute-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] set objName [java::new javax.management.Object Name $objNameString] $AdminControl getAttribute_jmx $objName processType` Using Jython: `objNameString = AdminControl.complete ObjectName('WebSphere: type=Server,*') import javax. management as mgmt objName = mgmt.Object Name(objNameString) AdminControl.getAttribute _jmx(objName, 'processType')`
getAttributes	Returns the attribute values for the names that you provide.	• Parameters using Jacl: name-String; attributes-java.lang.String • Parameters using Jython: name-String; attributes-java.lang.String or name-String; attributes-java.lang.Object[] • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] $AdminControl getAttributes $objName String "cellName nodeName"` Using Jython with string attributes: `objNameString = Admin Control.completeObject name('WebSphere:type =Server,*) AdminControl.getAttributes (objNameString, '[cellName nodeName]')` Using Jython with object attributes: `objNameString = Admin Control.completeObject name('WebSphere:type =Server,*) AdminControl.get Attributes(objNameString, ['cellName', 'nodeName'])`

getAttributes _jmx	Returns the attribute values for the names that you provide.	• Parameters: name-ObjectName; attributes-java.lang.String[] • Returns: javax.management. AttributeList	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] set objName [$AdminControl makeObjectName $objectNameString] set attrs [java::new {String[]} 2 {cellName nodeName}] $AdminControl getAttributes _jmx $objName $attrs` Using Jython: `objectNameString = AdminControl.complete ObjectName('type=Server,*') objName = AdminControl. makeObjectName (objectNameString) attrs = ['cellName', 'nodeName'] AdminControl.getAttributes _jmx(objName, attrs)`
getCell	Returns the name of the connected cell.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getCell` Using Jython: `AdminControl.getCell()` Example output: `Mycell`
getConfigId	Creates a configuration ID from an ObjectName or an ObjectName fragment. Use this ID with the `$AdminConfig` command. Not all MBeans that run have configuration objects that correspond. If several MBeans correspond to an ObjectName fragment, a warning is created and a configuration ID builds for the first MBean it finds.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set threadpoolCID [$AdminControl getConfigId node=mynode, type=ThreadPool,*]` Using Jython: `threadpoolCID = AdminControl. getConfigId('node=mynode, type=ThreadPool,*')`

getDefault Domain	Returns the default domain name from the server.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getDefaultDomain` Using Jython: `AdminControl.getDefaultDomain()` Example output: `WebSphere`
getDomain Name	Returns the domain name from the server.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getDomainName` Using Jython: `AdminControl.getDomainName()` Example output: `WebSphere`
getHost	Returns the name of your host.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getHost` Using Jython: `AdminControl.getHost()` Example output: `myhost`
getMBean Count	Returns the number of MBeans that are registered in the server.	• Parameters: None • Returns: java.lang.Integer	Example usage: Using Jacl: `$AdminControl getMBeanCount` Using Jython: `AdminControl.getMBeanCount()` Example output: `114`

getMBeanInfo _jmx	Returns 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** command.	• Parameters: name-ObjectName • Returns: javax.management. MBeanInfo	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server,*] set objName [$AdminControl makeObjectName $objectNameString] $AdminControl getMBeanInfo _jmx $objName` Using Jython: `objectNameString = AdminControl.complete ObjectName('type=Server,*') objName = AdminControl. makeObjectName (objectNameString) AdminControl.getMBeanInfo _jmx(objName)` Example output: `javax.management.modelmbean. ModelMBeanInfoSupport@ 10dd5f35`
getNode	Returns the name of the connected node.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getNode` Using Jython: `AdminControl.getNode()` Example output: `Myhost`
getPort	Returns the name of your port.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getPort` Using Jython: `AdminControl.getPort()` Example output: `8877`

getProperties ForData Source	Deprecated, no replacement. This command incorrectly assumes the availability of a configuration service when running in connected mode.	• Parameters: configId-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set ds [lindex [$AdminConfig list DataSource] 0] $AdminControl getProper tiesForDataSource $ds` Using Jython: `ds = AdminConfig.list ('DataSource')` `# get line separator import java.lang. System as sys lineSeparator = sys. getProperty('line. separator')` `dsArray = ds.split (lineSeparator) AdminControl.getProper tiesForDataSource (dsArray[0])` Example output: `WASX7389E: Operation not supported - get PropertiesForDataSource command is not supported.`
getType	Returns the connection type.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getType` Using Jython: `AdminControl.getType()` Example output: `SOAP`

help	Returns general help text for the AdminControl object.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl help` Using Jython: `AdminControl.help()` Example output: `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 a template name getAttribute_jmx 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_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

			Show help information invoke_jmx Given ObjectName, name of command, array of parameters and signature, invoke command on MBean specified 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 server setAttribute_jmx 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_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

help	Returns help text for the specific command of the AdminControl object. The command name is not case sensitive.	• Parameters: command-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl help getAttribute` Using Jython: `AdminControl.help('getAttribute')` Example output: `WASX7043I: command: getAttribute` `Arguments: object name, attribute` `Description: Returns value of` `"attribute" for the MBean` `described by "object name."`
invoke	Invokes the object operation without any parameter. Returns the result of the invocation.	• Parameters: name-java.lang.String; operationName-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl` `completeObjectName WebSphere:` `type=Server,*]` `$AdminControl invoke` `$objNameString stop` Using Jython: `objNameString = AdminControl.` `completeObjectName('WebSphere:` `type=Server,*')` `AdminControl.invoke(objNameString,` `'stop')`

invoke	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.	• Parameters: name-java.lang.String; operationName-java.lang.String; params-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]` `$AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled` Using Jython: `objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')` `AdminControl.invoke(objNameString, 'appendTraceString', 'com.ibm.*=all=enabled')`

invoke	Invokes the object operation by conforming the parameter list to the signature. Returns the result of the invocation.	• Parameters: name-java.lang.String; operationName-java.lang.String; params-java.lang.String; sigs-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere: type=Server,*] $AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled java.lang.String` Using Jython: `objNameString = AdminControl. completeObjectName('WebSphere: type=Server,*') AdminControl.invoke (objNameString, 'appendTrace String', 'com.ibm.*=all=enabled', 'java.lang.String')`
invoke_jmx	Invokes the object operation by conforming the parameter list to the signature. Returns the result of the invocation.	• Parameters: name-ObjectName; operationName-java.lang.String; params-java.lang.Object[]; signature-java.lang.String[] • Returns: java.lang.Object	Example usage: `set objNameString [$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'] AdminControl.invoke_jmx (objName, 'appendTraceString', parms, signature)`

isRegistered	If the ObjectName value is registered in the server, then the value is `true`.	• Parameters: name-java.lang.String • Returns: Boolean	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere: type=Server,*] $AdminControl isRegistered $objNameString` Using Jython: `objNameString = AdminControl. completeObjectName('WebSphere: type=Server,*') AdminControl.isRegistered (objNameString)`
isRegistered _jmx	If the ObjectName value is registered in the server, then the value is `true`.	• Parameters: name-ObjectName • Returns: Boolean	Example usage: Using Jacl: `set objectNameString [$AdminControl completeObjectName type=Server,*] set objName [$AdminControl makeObjectName $objNameString] $AdminControl isRegistered_jmx $objName` Using Jython: `objectNameString = AdminControl. completeObjectName('type=Server,*') objName = AdminControl. makeObjectName(objectNameString) AdminControl.isRegistered_jmx (objName)`

makeObject Name	A convenience command that creates 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.	• Parameters: name-java.lang.String • Returns: javax.management. ObjectName	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server, node=mynode,*] set objName [$AdminControl makeObjectName $objNameString]` Using Jython: `objectNameString = AdminControl.completeObjec tName('type=Server, node=mynode,*') objName = AdminControl. makeObjectName (objectNameString)`
queryNames	Returns a string that lists all the ObjectName objects based on the name template.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl queryNames WebSphere:type=Server,*` Using Jython: `AdminControl.queryNames ('WebSphere:type=Server,*')` Example output: `WebSphere:cell=Base ApplicationServerCell, name=server1,mbeanIdentifier= server1,type=Server,node= mynode,process=server1`

queryNames _jmx	Returns a set of ObjectName objects that are based on the ObjectName object and the QueryExp query that you provide.	• Parameters: name-javax.management. ObjectName;query-javax .management.QueryExp • Returns: java.util.Set	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server,*] set objName [$AdminControl makeObjectName $objNameString] set null [java::null] $AdminControl queryNames _jmx $objName $null` Using Jython: `objectNameString = AdminControl.completeObject Name('type=Server,*') objName = AdminControl.make ObjectName(objectNameString) AdminControl.queryNames_ jmx(objName, None)` Example output: `[WebSphere:cell=Base ApplicationServerCell, name=server1,mbeanIdentifier= server1,type=Server,node= mynode,process=server1]`
reconnect	Reconnects to the server, and clears information out of the local cache.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminControl reconnect` Using Jython: `AdminControl.reconnect()` Example output: `WASX7074I: Reconnect of SOAP connector to host myhost completed.`
setAttribute	Sets the attribute value for the name that you provide.	• Parameters: name-java.lang.String; attributeName- java.lang.String; attributeValue-java.lang.String • Returns: None	Example usage: 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,*') AdminControl.setAttribute (objNameString, 'trace Specification', 'com.ibm. *=all=disabled')`

setAttribute _jmx	Sets the attribute value for the name that you provide.	• Parameters: name-ObjectName; attribute-javax. management. Attribute • Returns: None	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName WebSphere: type=TraceService,*] set objName [$AdminControl makeObjectName $objectNameString] set attr [java::new javax. management.Attribute traceSpecification com.ibm. *=all=disabled] $AdminControl setAttribute_ jmx $objName $attr` Using Jython: `objectNameString = AdminControl. completeObjectName('WebSphere: type=TraceService,*') import javax.management as mgmt objName = AdminControl. makeObjectName(objectNameString) attr = mgmt.Attribute ('traceSpecification', 'com.ibm.*=all=disabled') AdminControl.setAttribute_ jmx(objName, attr)`
setAttributes	Sets the attribute values for the names that you provide and returns a list of successfully set names.	• Parameters using Jacl: name-String; attributes-java.lang.String • Parameters using Jython: name-String; attributes-java.lang.String or name-String; attributes-java.lang.Object[] • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere: type=TracesService,*] $AdminControl setAttributes $objNameString {{trace Specification com.ibm.ws. *=all=enabled}}` Using Jython with string attributes: `objNameString = AdminControl. completeObjectName('WebSphere: type=TracesService,*') AdminControl.setAttributes (objNameString, '[[trace Specification "com.ibm.ws. *=all=enabled"]]')` Using Jython with object attributes: `objNameString = AdminControl. completeObjectName('WebSphere: type=TracesService,*') 473 AdminControl.setAttributes (objNameString, [['trace Specification', 'com.ibm.ws. *=all=enabled']])`

setAttributes _jmx	Sets the attribute values for the names that you provide and returns a list of successfully set names.	• Parameters: name-ObjectName; attributes-javax.management. AttributeList • Returns: javax.management. AttributeList	Example usage: Using Jacl: `set objectNameString [$AdminControl completeObject Name WebSphere:type= TraceService,*] set objName [$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 $alist` Using Jython: `objectNameString = AdminControl.completeObject Name('WebSphere:type= TraceService,*') import javax.management as mgmt objName = AdminControl. makeObjectName(object NameString) attr = mgmt.Attribute ('traceSpecification', 'com.ibm.ws.*=all=enabled') alist = mgmt.AttributeList() alist.add(attr) AdminControl.setAttributes_ jmx(objName, alist)`
startServer	Starts the specified application server by locating it in the configuration. This command uses the default wait time. You can only use this command if the scripting client is connected to a node agent. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1` Using Jython: `AdminControl.startServer ('server1')`

startServer	Starts the specified application server by locating it in the configuration. The start process waits the number of seconds specified by the wait time for the server to start. You can only use this command if the scripting client is connected to a node agent. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, wait time-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 100` Using Jython: `AdminControl.startServer ('server1', 100)`
startServer	Starts the specified application server by locating it in the configuration. This command uses the default wait time. You can use this command when the scripting client is either connected to a node agent or to a deployment manager process. It returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 myNode` Using Jython: `AdminControl.startServer ('server1', 'myNode')`

startServer	Starts the specified application server by locating it in the configuration. The start process waits the number of seconds specified by the wait time for the server to start. You can use this command when the scripting client is either connected to a node agent or to a deployment manager process. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String, wait time-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 myNode 100` Using Jython: `AdminControl.startServer ('server1', 'myNode', 100)`
stopServer	Stops the specified application server. The command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1` Using Jython: `AdminControl.stopServer ('server1')`
stopServer	Stops the specified application server. If you set the flag to `immediate`, the server stops immediately. Otherwise, a normal stop occurs. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, immediate flag-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 immediate` Using Jython: `AdminControl.stopServer ('server1', 'immediate')`

stopServer	Stops the specified application server. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 myNode` Using Jython: `AdminControl.stopServer ('server1', 'my Node')`
stopServer	Stops the specified application server. If you set the flag to `immediate`, the server stops immediately. Otherwise, a normal stop occurs. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String, immediate flag-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 myNode immediate` Using Jython: `AdminControl.stopServer ('server1', 'my Node', 'immediate')`

test Connection	A convenience command communicates with the DataSource CfgHelper MBean to test a DataSource connection. This command works with the DataSource that resides in the configuration repository. If the DataSource to be tested is in the temporary workspace that holds the update to the repository, you have to save the update to the configuration repository before running this command. Use this command with the configuration ID that corresponds to the DataSource and the WAS40DataSource object types.	• Parameters: configId-java.lang.String • Returns: java.lang.String	Example usage: 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]` `AdminControl.testConnection(ds)` Example output: `WASX7217I: Connection to provided datasource was successful.`
	The return value is a message that contains the message indicating a successful connection or a connection with warning. If the connection fails, an exception is created from the server indicating the error.		

test Connection	Deprecated. This command can give false results and does not work when connected to a node agent. As of V5.0.2, the preferred way to test a data source connection is with the **test Connection** command that passes in the DataSource configId parameter as the only parameter.	• Parameters: configId-java.lang.String; props-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set ds [lindex [$AdminConfig list DataSource] 0]` `$AdminControl testConnection $ds {{prop1 val1}}` Using Jython: `# get line separator` `import java.lang.System as sys` `lineSeparator = sys.getProperty ('line.separator')` `ds = AdminConfig.list ('DataSource').split (lineSeparator)[0]` `AdminControl.testConnection(ds, '[[prop1 val1]]')` Example output: `WASX7390E: Operation not supported - testConnection command with config id and properties arguments is not supported. Use testConnection command with config id argument only.`
trace	Sets the trace specification for the scripting process to the value that you specify.	• Parameters: traceSpec-java.lang.String • Returns: None	Example usage: Using Jacl: `$AdminControl trace com.ibm.ws.scripting.*=all=enabled` Using Jython: `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:

Command name:	Description:	Parameters and return values:	Examples:
deleteUser AndGroup Entries	Deletes users or groups for all roles, and deletes user IDs and passwords for all of the RunAs roles that are defined in the application.	• Parameters: appname • Returns: None	Example usage: Using Jacl: `$AdminApp deleteUserAndGroup Entries myapp` Using Jython: `AdminApp.deleteUserAndGroup Entries('myapp')`
edit	Edits an application or module in non-interactive mode. The **edit** command changes the application deployment. Specify these changes in the options parameter. No options are required for the **edit** command.	• Parameters using Jacl: appname - string; options - string • Parameters using Jython: appname - string; options - string or appname - string; options - Jython list • Returns: string	Example usage: Using Jacl: `$AdminApp edit "JavaMail Sample" {-MapWebModToVH {{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}}}` Using Jython with string options: `AdminApp.edit("JavaMail Sample", '[-MapWebModToVH [["JavaMail 32 Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH]]]')` Using Jython with list options: `option = [["JavaMail 32 Sample WebApp", "mtcomps.war,WEB-INF/web.xml", "newVH"]]` `mapVHOption = ["-MapWebModToVH", option]` `AdminApp.edit("JavaMail Sample", mapVHOption)`
editInter active	Edits 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.	• Parameters using Jacl: appname - string; options - string • Parameters using Jython: appname - string; options - string or appname - string; options - Jython list • Returns: string	Example usage: Using Jacl: `$AdminApp editInteractive ivtApp` Using Jython: `AdminApp.editInteractive('ivtApp')`

export	Exports the application appname parameter to a file that you specify by file name.	• Parameters: appname, filename • Returns: None	Example usage: Using Jacl: `$AdminApp export "My App" /usr/me/myapp.ear` Using Jython: `AdminApp.export("My App", '/usr/me/myapp.ear')`
exportDDL	Extracts the data definition language (DDL) from the application appname parameter to the directoryname parameter that a directory specifies. The options parameter is optional.	• Parameters: appname, directoryname, options • Returns: None	Example usage: Using Jacl: `$AdminApp exportDDL "My App" /usr/me/DDL {-ddlprefix myApp}` Using Jython: `AdminApp.exportDDL("My App", '/usr/me/DDL', '[-ddlprefix myApp]')`

help	Displays general help for the AdminApp object.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminApp help` Using Jython: `print AdminApp.help()` Example output: `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
help	Displays help for an AdminApp command or installation option.	• Parameters: operation name • Returns: none	Example usage: Using Jacl: `$AdminApp help uninstall` Using Jython: `print AdminApp.help('uninstall')` Example output: `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.`

install	Installs an application in non-interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.	• Parameters using Jacl: earfile- string; options- string • Parameters using Jython: earfile- string; options- string or earfile- string; options- Jython list • Returns: None	Example usage: Using Jacl: `$AdminApp install c:/apps/myapp.ear` Using Jython: `AdminApp.install ('c:/apps/myapp.ear')` Many options are available for this command. You can obtain a list of valid options for an Enterprise Archive (EAR) file with the following command: Using Jacl: `$AdminApp options myApp.ear` Using Jython: `AdminApp.options('myApp.ear')` You can also obtain help for each object with the following command: Using Jacl: `$AdminApp help MapModulesToServers` Using Jython: `AdminApp.help('MapModulesTo Servers')`
install Interactive	Installs an application in interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.	• Parameters using Jacl: earfile- string; options- string • Parameters using Jython: earfile- string; options- string or earfile- string; options- Jython list • Returns: None	Example usage: Using Jacl: `$AdminApp installInteractive c:/websphere/appserver/ installableApps/jmsample.ear` Using Jython: `AdminApp.installInteractive ('c:/websphere/appserver/ installableApps/jmsample.ear')`
list	Lists the applications that are installed in the configuration.	• Parameters: None • Returns: application names	Example usage: Using Jacl: `$AdminApp list` Using Jython: `print AdminApp.list()` Example output: `adminconsole DefaultApplication ivtApp`

listModules	Lists the modules in an application. The options parameter is optional. The valid option is -server. This option lists the application servers on which the modules are installed.	• Parameters: appname, options • Returns: modules in the application	Example usage: Using Jacl: `$AdminApp listModules ivtApp` Using Jython: `print AdminApp.listModules ('ivtApp')` Example output: `ivtApp#ivtEJB.jar+META-INF/ ejb-jar.xml ivtApp#ivt_app.war+WEB-INF /web.xml` This example is formed by the concatenation of appname, #, module URI, +, and DD URI. You can pass this string to the **edit** and **editInteractive** AdminApp commands.

options	Displays a list of options for installing an Enterprise Archive (EAR) file.	• Parameters: earfile • Returns: Information about the valid installation options for an Enterprise Archive (EAR) file.	Example usage: Using Jacl: `$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear` Using Jython: `AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear')` Example usage: `WASX7112I: The following options are valid for "c:/websphere/appserver/installableapps/ivtApp.ear"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `EnsureMethodProtectionFor10EJB` `GetServerName` `preCompileJSPs` `nopreCompileJSPs` `distributeApp` `nodistributeApp` `useMetaDataFromBinary` `nouseMetaDataFromBinary` `deployejb` `nodeployejb` `createMBeansForResources` `nocreateMBeansForResources` `reloadEnabled` `noreloadEnabled` `deployws` `nodeployws` `usedefaultbindings` `defaultbinding.force` `allowPermInFilterPolicy` `noallowPermInFilterPolicy` `verbose` `update` `update.ignore.old` `update.ignore.new` `installed.ear.destination` `appname` `reloadInterval` `validateinstall` `deployejb.rmic` `deployejb.dbtype` `deployejb.dbschema` `deployejb.classpath` `deployws.classpath` `deployws.jardirs` `defaultbinding.datasource.jndi` `defaultbinding.datasource.username` `defaultbinding.datasource.password`

			defaultbinding.cf.jndi defaultbinding.cf.resauth defaultbinding.ejbjndi.prefix defaultbinding.virtual.host defaultbinding.strategy.file server node cell cluster contextroot custom
options	Displays a list of options for editing an existing application.	• Parameters: Application name • Returns: Information about the valid edit options for an application.	Example usage: Using Jacl: `$AdminApp options ivtApp` Using Jython: `AdminApp.options('ivtApp')` Example output: `WASX7112I: The following options are valid for "ivtApp"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `distributeApp` `nodistributeApp` `useMetaDataFromBinary` `nouseMetaDataFromBinary` `createMBeansForResources` `nocreateMBeansForResources` `reloadEnabled` `noreloadEnabled` `verbose` `installed.ear.destination` `reloadInterval`
options	Displays a list of options for editing a module in an existing application.	• Parameters: application module name. This parameter requires the same module name format as the output that is returned by the **listModules** command. • Returns: Information about the valid edit options for a module.	Example usage: Using Jacl: `$AdminApp options ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml` Using Jython: `AdminApp.options('ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml')` Example output: `WASX7112I: The following options are valid for "ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml"` `MapRolesToUsers` `BindJndiForEJBNon MessageBinding` `MapModulesToServers` `verbose`

options	Displays a list of options for installing or updating an application or application module file.	• Parameters: file, operation - The following list includes the valid values: – installapp - Installing the file that is specified – updateapp - Updating an existing application with the file that is specified – addmodule - Adding the module file that is specified to an existing application – updatemodule - Updating an existing module in an application with the module file that is specified • Returns: Information about the valid options that are available for the operation that is requested with the input file.	Example using the updateapp operation: Using Jacl: `$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear updateapp` Using Jython: `AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear', 'updateapp')` Example using the addmodule operation: Using Jacl: `$AdminApp options myModule.jar addmodule` Using Jython: `AdminApp.options('Default WebApplication.war', 'addmodule')` Example output using the updateapp operation: `WASX7112I: The following options are valid for "c:/websphere/appserver/installableApps/ivtApp.ear"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `EnsureMethodProtectionFor10EJB` `GetServerName` `preCompileJSPs` `nopreCompileJSPs` `distributeApp` `nodistributeApp` `useMetaDataFromBinary` `nouseMetaDataFromBinary` `deployejb` `nodeployejb` `createMBeansForResources` `nocreateMBeansForResources` `reloadEnabled` `noreloadEnabled` `deployws` `nodeployws` `usedefaultbindings` `defaultbinding.force` `allowPermInFilterPolicy` `noallowPermInFilterPolicy` `verbose` `update` `update.ignore.old` `update.ignore.new` `installed.ear.destination` `reloadInterval`

			deployejb.rmic deployejb.dbtype deployejb.dbschema deployejb.classpath deployws.classpath deployws.jardirs defaultbinding.datasource. jndi defaultbinding.datasource. username defaultbinding.datasource. password defaultbinding.cf.jndi defaultbinding.cf.resauth defaultbinding.ejbjndi.prefix defaultbinding.virtual.host defaultbinding.strategy.file appname contextroot custom contenturi contents operation Example output using the addmodule operation: WASX7112I: The following options are valid for "DefaultWebApplication.war" MapRolesToUsers MapEJBRefToEJB MapWebModToVH MapModulesToServers GetServerName preCompileJSPs nopreCompileJSPs deployejb nodeployejb deployws nodeployws usedefaultbindings defaultbinding.force verbose defaultbinding.datasource. jndi defaultbinding.datasource. username defaultbinding.datasource. password defaultbinding.cf.jndi defaultbinding.cf.resauth defaultbinding.ejbjndi.prefix defaultbinding.virtual.host defaultbinding.strategy.file server node cell cluster contextroot custom contenturi contents operation

publish WSDL	Publishes Web Services Description Language (WSDL) files for the application that is specified in the appname parameter to the file that is specified in the filename parameter.	• Parameters: appname, filename • Returns: None	Example usage: Using Jacl: `$AdminApp publishWSDL JAXRPCHandlerServer c:/temp/a.zip` Using Jython: `AdminApp.publishWSDL ('JAXRPCHandlerServer', 'c:/temp/a.zip')`
publish WSDL	Publishes Web Services Description Language (WSDL) files for the application that is specified in the appname parameter to the file that is specified in the filename parameter using the SOAP address prefixes that are specified in the soapAddress Prefixes parameter.	• Parameters: appname, filename, soapAddressPrefixes • Returns: None	Example usage: Using Jacl: `$AdminApp publishWSDL JAXRPCHandlersServer c:/temp/a.zip {{JAXRPCHandlersServerApp.war {{http http://localhost:9080}}}}` Using Jython: `AdminApp.publishWSDL ('JAXRPCHandlersServer', 'c:/temp/a.zip', '[[JAXRPCHandlersServerApp.war [[http http://localhost:9080]]]]')`
searchJNDI References	Lists applications that refer to the Java Naming and Directory Interface (JNDI) name on a specific node.	• Parameters: Node configuration ID, options • Returns: string	Example usage: 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.searchJNDI References(node, '[-JNDIName eis/J2CCF1 -verbose]')` Example 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]`

taskInfo	Provides information about a particular task option for an application file.	• Parameters: earfile, task name • Returns: None	Example usage: Using Jacl: `$AdminApp taskInfo c:/websphere/appserver/installableApps/jmsample.ear MapWebModToVH` Using Jython: `print AdminApp.taskInfo ('c:/websphere/appserver/installableApps/jmsample.ear', 'MapWebModToVH')` Example 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`

uninstall	Uninstalls an existing application.	• Parameters: appname- string • Returns: None	Example usage: Using Jacl: `$AdminApp uninstall myApp` Using Jython: `AdminApp.uninstall('myApp')` Example 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.`

update Access IDs	Updates the access ID information for users and groups that are assigned to various roles that are defined in the application. The access IDs are read from the user registry and saved in the application bindings. This operation improves run-time performance of the application. Call 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 is set to `NONE`. You must be connected to a server to invoke this command.	• Parameters: appname, bALL • Returns: None	Example usage: Using Jacl: `$AdminApp updateAccessIDs myapp true` Using Jython: `AdminApp.updateAccessIDs ('myapp', 'true')`
	The bALL Boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` if you want to retrieve access IDs for users or groups that do not have an access ID in the application bindings.		

view	View the task that is specified by the taskname option parameter for the application or by the module that is specified by the 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.	• Parameters: name, taskname option • Returns: string	Example usage: Using Jacl: `$AdminApp view adminconsole {-tasknames}` Using Jython: `AdminApp.view('adminconsole', ['-tasknames'])` Example output: `MapModulesToServers` `MapWebModToVH` `MapRolesToUsers` Using Jacl: `$AdminApp view adminconsole {-MapModulesToServers}` Using Jython: `AdminApp.view('adminconsole', ['-MapModulesToServers'])` Example output: `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=juniarti Network,node=juniartiManager, server=dmgr` Example usage: Using Jacl: `$AdminApp view adminconsole# adminconsole.war+WEB-INF/web.xml {-MapRolesToUsers}` Using Jython: `AdminApp.view('adminconsole# adminconsole.war+WEB-INF/web.xml', ['-MapRolesToUsers'])` Example output: `MapRolesToUsers: Mapping Users to Roles` `Each role that is defined in the application or the module must be mapped to a user or a group from the user registry of the domain:`

			Role: administrator Everyone?: No All Authenticated?: No Mapped Users: Mapped Groups: Role: operator Everyone?: No All Authenticated?: No Mapped Users: Mapped Groups: Role: configurator Everyone?: No All Authenticated?: No Mapped Users: Mapped Groups: Role: monitor Everyone?: No All Authenticated?: No Mapped Users: Mapped Groups:

update	Updates an application in non-interactive mode. Provide the application name, content type, and update options.	• Parameters using Jacl: appname, content type, options – string format • Parameters using Jython: appname, content type, option- string or list format • Returns: String This command supports the addition, removal, and update of application subcomponents or the entire application. 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.	Example usage: Using Jacl: `$AdminApp update myApp file {-operation add -contents c:/apps/myApp/web.xml -contenturi META-INF/web.xml}` Using Jython with string options: `AdminApp.update('myApp', 'file', '[-operation add -contents c:/apps/myApp/web.xml -contenturi META-INF/web.xml]')` Using Jython with list options: `AdminApp.update('myApp', 'file', ['-operation', 'add', '-contents', 'c:/apps/myApp/web.xml', '-contenturi', 'META-INF/web.xml'])` Example 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:`

		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	`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\Scriptfb5a191` `b4e\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.`

		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. For a list of the valid options for the **update** command, see "Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands" on page 307.	

update Interactive	Updates an application in interactive mode. Provide the application name, content type, and update options.	• Parameters using Jacl: appname, content type, options - string format • Parameters using Jython: appname, content type, option - string or list format • Returns: String 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. Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list contains 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 **install** command.	Example usage: Using Jacl: `$AdminApp updateInteractive myApp modulefile {-operation add -contents c:/apps/myApp/Increment.jar -contenturi Increment.jar -nodeployejb -BindJndiForEJBNonMessageBinding {{"Increment Enterprise JavaBeans" Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}` Using Jython string: `AdminApp.updateInteractive ('myApp', 'modulefile', '[-operation add -contents c:/apps/myApp/Increment.jar -contenturi Increment.jar -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment Enterprise JavaBeans" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')` Using Jython list: `bindJndiForEJBValue = [["Increment Enterprise JavaBeans", "Increment", "Increment.jar,META-INF/ejb-jar.xml", "Inc"]]` `AdminApp.updateInteractive ('myApp', 'modulefile', ['-operation', 'add', '-contents', 'c:/apps/myApp/Increment.jar', '-contenturi', 'Increment.jar', '-nodeployejb', '-BindJndiForEJBNon MessageBinding', bindJndiForEJBValue])` Example 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 Enterprise Java Bean EJB: Increment URI: Increment.jar, META-INF/ejb-jar.xml JNDI Name: [Inc]:`

		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.	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 Enterprise Java Bean 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 Enterprise Java Bean 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 Enterprise Java Bean URI: Increment.jar,META-INF /ejb-jar.xml Server: [WebSphere:cell= myCell,node=myNode, server=server1]:

		• 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, additional 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 the 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 subset of application components provided in a zip file format you can update, add, and delete files and modules.	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 Enterprise Java Bean 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 Enterprise Java Bean 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,]

		The zip file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, this file contains application artifacts in the same hierarchical structure as they are displayed 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, the binding information and application options cannot be specified and are ignored, if provided. For a list of the valid options for the **updateInteractive** command, see "Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands" on page 307.	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\Scriptfb5 a487089\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\Scriptfb5 a487089\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.

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. The options listed in this article apply to all of these commands except where noted.

See Commands for the AdminApp object for more detailed information on how to use the commands. See Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands for a list of applicable commands for each option.

The following options are available for the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands:

Option name:	Description:	Examples:

ActSpecJNDI	Binds J2C activation specs to destination JNDI names. You can bind J2C activation specs in your application or module to a destination JNDI name. This option is optional. Each element of the ActSpecJNDI option consists of the following fields: RARModule, uri, j2cid, j2c.jndiName. j2c.jndiName field, can be assigned a value. The current contents of the option after running default bindings include: • RARModule: <rar module name> • uri: <rar name>,META-INF/ra.xml • j2cid: <messageListenerType> • j2c.jndiName: null • Object identifier: javax.jms.MessageListener	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 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']]])`

	You can only use this option if the activation spec has the `Destination` property defined in the ra.xml file and the introspected type of the `Destination` property is the following: `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 require an update.	
`allowPermInFilterPolicy`	Specifies to 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.	
appname	Specifies the name of the application. The default is the display name of the application.	

BackendId Selection	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.	Using Jacl: `$AdminApp install c:/myapp.ear {-BackendIdSelection {{Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml \DB2UDBNT_V72_1}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-BackendIdSelection [[Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml DB2UDBNT_V72_1]]]')`
BindJndiFor EJBMessage Binding	Binds enterprise beans to listener port names or Java Naming and Directory Interface (JNDI) names. Use this option to provide missing data or update a task. Ensure each message-driven enterprise bean in your application or module is bound to a listener port name. Each element of the BindJndiForEJBMessageBinding option consists of the following fields: EJBModule, EJB, uri, listenerPort, JNDI, jndi.dest, and actspec.auth. Some of these fields, can be assigned values: listenerPort, JNDI, jndi.dest, and actspec.auth.	Using Jacl: `$AdminApp install $ear {-BindJndiForEJBMessage Binding {{Ejb1 MessageBean ejb-jar-ic.jar,META-INF/ejb-jar.xml myListener Port jndi1 jndiDest1 actSpecAuth1}}}` Using Jython: `AdminApp.install(ear, ['-BindJndiForEJBMessage Binding', [['Ejb1', 'MessageBean', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'myListener Port', 'jndi1', 'jndiDest1', 'actSpecAuth1']]])`

	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 • listenerPort: MessageBeanPort • JNDI: null • jndi.dest: null • actspec.auth: null 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.	

BindJndiFor EJBNon Message Binding	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. 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.	Using Jacl: `$AdminApp install c:/myapp.ear {-BindJndiForEJBNon MessageBinding {{"Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-BindJndiFor EJBNonMessageBinding [["Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean]]]')`
cell	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.	

cluster	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.	
contents	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	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	Specifies the context root that you use when installing a stand-alone Web archive (WAR) file.	
CorrectOracle IsolationLevel	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-CorrectOracleIsolationLevel {{AsyncSender jms/MyQueueConnection Factory jms/Resource1 2}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-CorrectOracleIsolationLevel [[AsyncSender jms/MyQueueConnection Factory jms/Resource1 2]]]')`

CorrectUse System Identity	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-CorrectUseSystem Identity {{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: `AdminApp.install('c:/myapp.ear', '[-CorrectUse SystemIdentity [[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]]]')`
createMBeans ForResources	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.	

custom	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.	

DataSource For10CMP Beans	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 DataSourceFor 10CMPBeans 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:	Using Jacl: `$AdminApp install c:/app1.ear {-DataSource For10CMPBeans {{"Increment CMP 1.1 EJB" IncCMP11 IncCMP11.jar,META-INF/ejb-jar.xml myJNDI user1 password1 loginName1 authProps1}}}` Using Jython: `AdminApp.install('c:/app1.ear', ['-Data SourceFor10CMPBeans', [["Increment CMP 1.1 EJB", 'IncCMP11', 'IncCMP11.jar, META-INF/ejb-jar.xml', 'myJNDI', 'user1', 'password1', 'loginName1', 'authProps1']]])`

	• EJBModule: Increment CMP 1.1 EJB • EJB: IncCMP11 • uri: IncCMP11.jar,META-INF/ejb-jar.xml • JNDI: DefaultDatasource • userName: null • password: null • login.config.name: DefaultPrincipalMapping • auth.props: • LoginConfiguration: Name • Properties 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 require an update.	

DataSource For20CMP Beans	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 DataSourceFor 20CMPBeans 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:	Using Jacl: `$AdminApp install c:/app1.ear {-DataSource For20CMPBeans {{"Increment Enterprise Java Bean" Increment Increment.jar, META-INF/ejb-jar.xml jndi1 container}}}` Using Jython: `AdminApp.install('c:/app1.ear', ['-DataSourceFor20CMPBeans', [["Increment Enterprise Java Bean", 'Increment', 'Increment.jar, META-INF/ejb-jar.xml', 'jndi1', 'container']]])`

	• EJBModule: Increment enterprise bean • EJB: Increment • uri: Increment.jar,META-INF/ejb-jar.xml • JNDI: DefaultDatasource • resAuth: cmpBinding.perConnectionFactory • login.config.name: DefaultPrincipalMapping • auth.props: • LoginConfiguration: Name • Properties 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 needed for your application. You only need to provide data for rows or entries that are missing information, or require an update.	

DataSource For10EJB Modules	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 DataSourceFor 10EJBModules 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:	Using Jacl: `$AdminApp install c:/app1.ear {-DataSource For10EJBModules {{"Increment CMP 1.1 EJB" IncCMP11.jar,META-INF/ejb-jar.xml yourJNDI user2 password2 loginName authProps}}}` Using Jython: `AdminApp.install('c:/app1.ear', ['-DataSource For10EJBModules', [["Increment CMP 1.1 EJB", 'IncCMP11.jar,META-INF/ejb-jar.xml', 'yourJNDI', 'user2', 'password2', 'loginName', 'authProps']]])`

	• EJBModule: Increment CMP 1.1 enterprise bean • uri: IncCMP11.jar,META-INF/ejb-jar.xml • JNDI: DefaultDatasource • userName: null • password: null • login.config.name: DefaultPrincipalMapping • auth.props: • LoginConfiguration: Name • Properties 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 an update.	

DataSource For20EJB Modules	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 DataSourceFor 20EJBModules 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, and auth.props. The current contents of the option after running default bindings include:	Using Jacl: `$AdminApp install c:/app1.ear {-DataSourceFor20EJBModules {{"Increment Enterprise Java Bean" Increment.jar,META-INF/ejb-jar.xml jndi2 container}}}` Using Jython: `AdminApp.install('c:/app1.ear', ['-DataSourceFor20EJBModules', [["Increment Enterprise Java Bean", 'Increment.jar,META-INF/ejb-jar.xml', 'jndi2', 'container']]])`

	• EJBModule: Increment enterprise bean • uri: Increment.jar,META-INF/ejb-jar.xml • JNDI: DefaultDatasource • resAuth: cmpBinding.perConnectionFactory • login.config.name: DefaultPrincipalMapping • auth.props: • LoginConfiguration: Name • Properties 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.	
defaultbinding. cf.jndi	Specifies the Java Naming and Directory Interface (JNDI) name for the default connection factory.	
defaultbinding. cf.resauth	Specifies the RESAUTH for the connection factory.	
defaultbinding. datasource.jndi	Specifies the Java Naming and Directory Interface (JNDI) name for the default data source.	

defaultbinding. datasource. password	Specifies the password for the default data source.	
defaultbinding. datasource. username	Specifies the user name for the default data source.	
defaultbinding. ejbjndi.prefix	Specifies the prefix for the enterprise bean Java Naming and Directory Interface (JNDI) name.	
defaultbinding. force	Specifies that the default bindings override the current bindings.	
defaultbinding. strategy.file	Specifies a custom default bindings strategy file.	
defaultbinding. virtual.host	Specifies the default name for a virtual host.	
depl.extension. reg	Deprecated. No replication option is available.	
deployejb	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	Specifies an extra class path for the EJBDeploy tool.	
deployejb. dbschema	Specifies the database schema for the EJBDeploy tool.	

deployejb. dbtype	Specifies the database type for the EJBDeploy tool. Possible values include: `CLOUDSCAPE_V5` `DB2UDB_V72` `DB2UDBOS390_V6` `DB2UDBISERIES` `INFORMIX_V73` `INFORMIX_V93` `MSSQLSERVER_V7` `MSSQLSERVER_2000` `ORACLE_V8` `ORACLE_V9I` `SYBASE_V1200` For a list of current supported database vendor types, run `ejbdeploy -?`.	
deployejb. rmic	Specifies extra RMIC options to use for the EJBDeploy tool.	
deployws	Specifies to deploy Web services during installation. This option does not require a value. The default value is: `nodeployws`.	
deployws. classpath	Specifies the extra class path to use when you deploy Web services.	
deployws. jardirs	Specifies the extra extension directories to use when you deploy Web services.	
distributeApp	Specifies that the application management component distributes application binaries. This option does not require a value. This setting is the default.	

EmbeddedRar	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 specs 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: `RARModule: <rar module name> uri: <rar name>, META-INF/ra.xml j2cid: <identifier of the J2C object> j2c.name: j2cid j2c.jndiName: eis/j2cid` Where j2cid is: `J2C connection factory : connection FactoryInterface J2C admin object :adminObject Interface J2C activation spec : message listener type`	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.FVTMessage Provider1} {"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml fvt.adapter. message.FVTMessageProvider2 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.FVTMessage Provider1', '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.FVTMessage Provider2']]])`  Using wsadmin.properties: `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.FVTMessage Provider1", "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.FVTMessage Provider2"]]]`

	If the ID is not unique in the ra.xml file, -<number> will be added. For example, javax.sql.DataSource-2. 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.	
EnsureMethod Protection For10EJB	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-EnsureMethod ProtectionFor10EJB {{"Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""} {"Timeout EJB Module" TimeoutEJBBean.jar,META-INF/ejb-jar.xml methodProtection.denyAllPermission}}}` Using Jython: `AdminApp.install('c:/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]]]')` 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.

EnsureMethod Protection For20EJB	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.	Using Jacl: `$AdminApp install c:/myapp.ear {-EnsureMethod ProtectionFor20EJB {{CustomerEjbJar customerEjb.jar,META-INF/ejb-jar.xml method Protection.uncheck} {SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-EnsureMethodProtectionFor20EJB [[CustmerEjbJar customerEjb.jar,META-INF/ejb-jar.xml methodProtection.uncheck] [SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml 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.
installdir	Deprecated. This option is replaced by the `installed.ear.destination` option.	
installed.ear. destination	Specifies the directory to place application binaries.	

MapModules ToServers	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-MapModulesTo Servers {{"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=server1}}}`  Using Jython: `AdminApp.install('c:/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=server1]]]')`

MapEJB RefToEJB	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. 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.	Using Jacl: `$AdminApp install c:/myapp.ear {-MapEJBRefToEJB {{"Examples Application" "" examples.war, WEB-INF/web.xml BeenThereBean com.ibm.websphere. beenthere.BeenThere IncBean}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-MapEJBRefToEJB [["Examples Application" "" examples.war,WEB-INF/web.xml BeenThereBean com.ibm.websphere. beenthere.BeenThere IncBean]]]')`

MapMessage Destination RefToEJB	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: • EJB Module: ejb-jar-ic.jar • EJB: MessageBean • URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml • JNDI Name: [eis/J2CACT1]: • Destination JNDI Name: [jms/TopicName]:	Using Jacl: `$AdminApp install $earfile {-MapMessageDestination RefToEJB {{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: `AdminApp.install(ear1, ['-MapMessageDestination RefToEJB', [['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']]])`

	• The default JNDI name will be picked up from the corresponding message destination 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 require an update.	
MapRes EnvRef ToRes	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-MapResEnvRefToRes {{AsyncSender AsyncSender asyncSenderEjb.jar, META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-MapResEnvRefToRes [[AsyncSender AsyncSender asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2]]]')`

MapRes RefTo EJB	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. Each element of the MapResRefToEJB option consists of the following fields: module, EJB, uri, referenceBinding, resRef.type, JNDI, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, login.config.name, auth.props. The JNDI field is required. The current contents of the option after running default bindings include:	Using Jacl: `$AdminApp install c:/app1.ear {-MapResRefToEJB {{deplmtest.jar MailEJBObject deplmtest.jar, META-INF/ejb-jar.xml mail/MailSession9 javax.mail.Session jndi1 login1 authProps1} {"JavaMail Sample WebApp" "" mtcomps.war, WEB-INF/web.xml mail/MailSession9 javax.mail. Session jndi2 login2 authProps2}}}` Using Jython: `AdminApp.install('c:/app1.ear', ['-MapResRefToEJB', [['deplmtest.jar', 'MailEJBObject', 'deplmtest.jar,META-INF/ ejb-jar.xml mail/MailSession9', 'javax.mail.Session', 'jndi1', 'login1', 'authProps1'], ["JavaMail Sample WebApp", "", 'mtcomps.war,WEB-INF/web.xml', 'mail/MailSession9', 'javax.mail. Session', 'jndi2', 'login2', 'authProps2']]])`

	• Module: deplmtest.jar • EJB: MailEJBObject • uri: deplmtest.jar,META-INF/ejb-jar.xml • referenceBinding: mail/MailSession9 • resRef.type: javax.mail.Session • JNDI: mail/DefaultMailSession • login.config.name: DefaultPrincipalMapping • auth.props: 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 an update.	

MapRoles ToUsers	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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-MapRolesToUsers {{"All Role" No Yes "" ""} {"Every Role" Yes No "" ""} {DenyAllRole No No user1 group1}}}` Using Jython: `AdminApp.install('c:/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

MapRun AsRoles ToUsers	Maps RunAs Roles to users. The enterprise beans you that 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 require an update.	Using Jacl: `$AdminApp install c:/myapp.ear {-MapRunAsRolesToUsers {{UserRole user1 password1} {AdminRole administrator administrator}}}` Using Jython: `AdminApp.install('c:/myapp.ear', '[-MapRunAsRolesToUsers [[UserRole user1 password1] [AdminRole administrator administrator]]]')`

MapWeb ModTo VH	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 require an update.	Using Jacl: `$AdminApp install c:/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: `AdminApp.install('c:/myapp.ear', '[-MapWebModToVH [["Default Application" default_app.war,WEB-INF/web.xml default_host] ["Examples Application" examples.war,WEB-INF/web.xml default_host]]]')`
noallow PermIn Filter Policy	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 it does not require a value.	

node	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.	
nocreate MBeans For Resources	Specifies that MBeans are not created for all resources such as, servlets, JSPs, 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.	
nodeploy ejb	Specifies not to run the EJBDeploy tool during installation. This option is the default setting and it does not require a value.	
nodeploy ws	Specifies not to deploy Web services during installation. This option is the default setting and it does not require a value.	

nodistribute App	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.	
noreload Enabled	Disables class reloading. This option does not require a value. The default setting is the `reloadEnabled` option.	
nopreCompile JSPs	Specifies not to precompile JavaServer Pages files. This option is the default setting and it does not require a value.	
noprocess Embedded Config	Use this option to 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.	

nouseMeta DataFrom Binary	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 it 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.	
nouse default bindings	Specifies not to use default bindings for installation. This option is the default setting and it does not require a value.	

operation	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`.	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 c:/apps/app1/my.xml -contenturi app1.jar/my.xml}` Using Jython string: `AdminApp.update('app1', 'file', '[-operation update -contents c:/apps/app1/my.xml -contenturi app1.jar/my.xml]')` Using Jython list: `AdminApp.update('app1', 'file', ['-operation', 'update', '-contents', 'c:/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.
process Embedded Config	Use this option to process 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.	

preCompile JSPs	Specifies to precompile the JavaServer Pages files. This option does not require a value. The default is nopreCompileJSPs.	
reload Enabled	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 it does not require a value.	
reload Interval	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.	
server	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.	

update	Updates the installed application with a new version of the enterprise archive file (EAR) file. This option does not require a value. The application that is being updated, 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	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	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.	

useMeta DataFrom Binary	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.	
usedefault bindings	Specifies to use default bindings for installation. This option does not require a value. The default setting is nousedefaultbindings.	

validate install	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	Causes additional messages to display during installation. This option does not require a value.	

WebServices ClientBind Deployed WSDL	The immutable values for this option identify 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.	Using Jacl: `$AdminApp install WebServicesSamples.ear {-WebServicesClientBindDeployedWSDL {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl}}}` Using Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindDeployedWSDL [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl]]]')`

	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`.	
WebServices ClientBind PortInfo	The immutable values identify 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.	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 Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPortInfo [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS]]]')`

WebServices ClientBind PreferredPort	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.	Using Jacl: `$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort}}}` Using Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPreferredPort [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort]]]')`

WebServices ServerBind Port	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).	Using Jacl: `$AdminApp install WebServicesSamples.ear {-WebServicesServerBindPort {{AddressBookW2JE.jar service/ WSLoggerService2 WSLoggerJMS {} Session}}}` Using Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesServerBindPort [[AddressBookW2JE.jar service/WSLoggerService2 WSLoggerJMS "" 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`.	

WebServices ClientCustom Property	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″}}	Using Jacl: `$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty {{AddressBookW2JE.jar AddressBookService AddressBook http.proxyHost +http.proxyPort myhost+80}}}` Using Jython: `AdminApp.edit ( 'WebServicesSamples', '[ -WebServicesServerCustomProperty [[AddressBookW2JE.jar AddressBookService AddressBook http.proxyHost+http.proxyPort myhost+80]]]')`

WebServices ServerCustom Property	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/value pairs: {{″n1″ ″v1″}, {″n2″ ″v2″}, {″n3″ ″ v3″}}	Using Jacl: `$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty {{AddressBookW2JE.jar AddressBook Service AddressBook com.ibm.websphere. webservices.http. responseContentEncoding deflate}}}` Using Jython: `AdminApp.edit ( 'WebServicesSamples', '[ -WebServicesServerCustomProperty [[AddressBookW2JE.jar AddressBookService AddressBook com.ibm.websphere.webservices.http. responseContentEncoding deflate]]]')`

Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands:

The following 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.

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
allowPermInFilterPolicy	Yes	Yes				
appname	Yes	Yes				
BackendIdSelection	Yes	Yes	Yes	Yes		
BindJndiForEJBMessagingBinding	Yes	Yes	Yes	Yes	Yes	Yes
BindJndiForEJBNonMessageBinding	Yes	Yes	Yes	Yes	Yes	Yes

cell	Yes	Yes	Yes			
cluster	Yes	Yes	Yes			
contents		Yes	Yes	Yes		
contenturi		Yes	Yes	Yes		
contextroot	Yes	Yes	Yes			
CorrectOracleIsolation Level	Yes	Yes	Yes	Yes	Yes	Yes
CorrectUseSystem Identity	Yes	Yes	Yes	Yes	Yes	Yes
createMBeansFor Resources	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		
defaultbinding. cf.jndi	Yes	Yes	Yes	Yes		
defaultbinding. cf.resauth	Yes	Yes	Yes	Yes		
defaultbinding. datasource. password	Yes	Yes	Yes	Yes		
defaultbinding. datasource. username	Yes	Yes	Yes	Yes		
defaultbinding. ejbjndi. prefix	Yes	Yes	Yes	Yes		
defaultbinding. force	Yes	Yes	Yes	Yes		
defaultbinding. strategy.file	Yes	Yes	Yes	Yes		
defaultbinding. virtual.host	Yes	Yes	Yes	Yes		
depl.extension. reg (deprecated)						
deployejb	Yes	Yes	Yes	Yes		
deployejb.classpath	Yes	Yes	Yes	Yes		
deployejb.dbschema	Yes	Yes	Yes	Yes		
deployejb.dbtype	Yes	Yes	Yes	Yes		
deployejb.rmic	Yes	Yes	Yes	Yes		
deployws	Yes	Yes	Yes	Yes		
deployws.classpath	Yes	Yes	Yes	Yes		
deployws.jardirs	Yes	Yes	Yes	Yes		
distributeApp	Yes	Yes			Yes	
EmbeddedRar	Yes	Yes	Yes	Yes	Yes	Yes
EnsureMethod ProtectionFor 10EJB	Yes	Yes	Yes	Yes		
EnsureMethod ProtentionFor 20EJB	Yes	Yes	Yes	Yes		
installdir (deprecated)						
installed.ear.destination	Yes	Yes			Yes	
MapMessage Destination RefToEJB	Yes	Yes	Yes	Yes	Yes	Yes

MapModulesToServers	Yes	Yes	Yes	Yes	Yes	Yes
MapEJBRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapResEnvRefToRes	Yes	Yes	Yes	Yes	Yes	Yes
MapResRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapRolesToUsers	Yes	Yes			Yes	Yes
MapRunAsRolesToUsers	Yes	Yes	Yes	Yes	Yes	Yes
MapWebModToVH	Yes	Yes	Yes	Yes	Yes	Yes
noallowPermIn FilterPolicy	Yes	Yes				
nocreateMBeans ForResources	Yes	Yes			Yes	
node	Yes	Yes	Yes			
nodeployejb	Yes	Yes	Yes	Yes		
nodeployws	Yes	Yes	Yes	Yes		
nodistributeApp	Yes	Yes			Yes	
nopreCompileJSPs	Yes	Yes	Yes	Yes		
noprocessEmbedded Config	Yes	Yes				
noreloadEnabled	Yes	Yes			Yes	
nousedefaultbindings	Yes	Yes	Yes	Yes		
nouseMetaData FromBinary	Yes	Yes			Yes	
operation		Yes	Yes	Yes		
preCompileJSPs	Yes	Yes	Yes	Yes		
processEmbedded Config	Yes	Yes				
reloadEnabled	Yes	Yes			Yes	
reloadInterval	Yes	Yes			Yes	
server	Yes	Yes	Yes			
update	Yes	Yes				
update.ignore.old	Yes	Yes		Yes		
update.ignore.new	Yes	Yes		Yes		
useMetaData FromBinary	Yes	Yes			Yes	
usedefaultbindings	Yes	Yes	Yes	Yes		
validateinstall	Yes				Yes	
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. For example:

```
WASX7278I: Generated command line: install c:/websphere/appserver/installableapps/
jmsample.ear {-BindJndiForEJBNonMessageBinding {{deplmtest.jar MailEJBObject
deplmtest.jar,META-INF/ejb-jar.xml ejb/JMSampEJB1 }} -MapResRefToEJB
{{deplmtest.jar MailEJBObject deplmtest.jar,META-INF/ejb-jar.xml
mail/MailSession9 javax.mail.Session mail/DefaultMailSessionX }
{"JavaMail Sample WebApp"  mtcomps.war,WEB-INF/web.xml mail/MailSession9
javax.mail.Session mail/DefaultMailSessionY }} -MapWebModToVH
{{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml newhost }}
-nopreCompileJSPs -novalidateApp -installed.ear.destination
c:/mylocation -distributeApp -nouseMetaDataFromBinary}
```

Commands for the AdminTask object

Use AdminTask object to run an administrative command. Administrative commands are discovered dynamically when you start the wsadmin tool. The administrative commands that are available for your use, and what you can do with them, depends on the edition of the WebSphere Application Server that you have.

You can start the scripting client without 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. 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 AdminTask object:

Comman d name:	Group name:	Descrip tion:	Target object:	Parameters and return values:	Examples:

addNode Group Member	Node Group Commands group	The **addNode Group Member** command adds a member to a node group. Nodes may be members of more than one node group. The command will do validity checking to ensure the following: • Distributed and z/OS nodes are not combined in the same node group.	The target object is the node group where the member will be created. This target object is required.	• Parameters: **- nodeName** The name of the node to be added to a node group. This parameter is required. • Returns: Node group member object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask addNodeGroup Member WBINodeGroup {-nodeName WBINode}` • Using Jython: `AdminTask.addNodeGroup Member('WBINodeGroup', '[-nodeName WBINode]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask addNodeGroup Member {-interactive}` • Using Jython: `AdminTask.addNodeGroup Member ('[-interactive]')`

addSIB WSInbo undPort	SIB Web Services group	The **addSIB WSInbound Port** command adds the configuration for an inbound port to an inbound service. This command will fail if: • the port name is already in use by another inbound port for the inbound service or the end point listener that you specified. • the template port that you specified does not exist in the template WSDL of the inbound service.	The object name of the inbound service to which the port will be added.	• Parameters: **name** The name of the port. (required) **endpointListener** The name of the associated end point listener. (required) **node** The node where the endpoint listener is located. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional) **server** The server where the endpoint listener is located. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional) **cluster** The cluster where the endpoint listener is located. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional)	**Batch mode example usage:** • Using Jacl: `set inPort [$AdminTask addSIBWSInboundPort $inService {-name "MyServiceSoap" -endpointListener "SOAPHTTP1" -node "MyNode" -server "server1"}]` • Using Jython: `inPort = AdminTask.addSIBWSInboundPort (inService, '[-name MyServiceSoap -endpointListener SOAPHTTP1 -node MyNode -server server1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask addSIBWSInboundPort {-interactive}` • Using Jython: `AdminTask.addSIBWSInboundPort ('[-interactive]')`
				templatePort The name of the port in the template WSDL to use as a basis for the binding of the port. (optional) • Returns: The object name of the inbound port object that was created.	

add SIB WS Outbound Port	SIB Web Services group	The **add SIBWS Outbound Port** command adds the configuration for an outbound port to an outbound service.	The object name of the outbound service for which the port will be associated.	• Parameters: **name** The name of the port in the WSDL of the service provider. (required) **node** Node where the port destination will be localized. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional) **server** The server where the port destination will be localized. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional) **cluster** The cluster where the port destination will be localized. You must specify the node parameter, the server parameter, or the cluster parameter. (conditional)	**Batch mode example usage:** • Using Jacl: `set outPort [$AdminTask addSIBWS OutboundPort $outService {-name "MyServiceSoap" -node "MyNode" -server "server"}]` • Using Jython: `outPort = AdminTask.addSIBWSOutboundPort (outService, '[-name MyServiceSoap -node MyNode -server server]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask addSIBWS OutboundPort {-interactive}` • Using Jython: `AdminTask.addSIBWS OutboundPort ('[-interactive]')`

				destination The name of the port destination. (optional) **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: The object name of the outbound port object that you created.	

addSI Bus Member	SIB Admin Commands group	Use this command to add a server or a cluster to a SIB bus.	None	• Parameters: **bus** name of bus to add member to (String, required) **node** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **server** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **cluster** to specify a cluster bus member, supply cluster name but not node and server name (String, optional) **createDefault Datasource** set this to true if a default data source should be created when the messaging engine is created. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask addSIBus Member {-bus` *`busname`* `-node` *`nodename`* `-server` *`servername`* `-description` *`text`*`}` • Using Jython: `AdminTask.addSIBus Member('[-bus` *`busname`* `-node` *`nodename`* `-server` *`servername`* `-description "`*`text`*`"]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask addSIBus Member {-interactive}` • Using Jython: `AdminTask.addSIBus Member ('[-interactive]')`
				datasourceJndi Name the JNDI name of the data source to be referenced from the datastore created when the member is added to the bus (String, optional)	

addWSGWTarget Service	WS Gateway group	The **add WSGW Target Service** command adds a target to a gateway service. You must specify the targetService parameter or the targetDestination parameter.	Object name of the GatewayService object	• Parameters: **name** The administrative name of the target service. (Required) **targetDestination** The name of the target destination. This can be within the same bus as the gateway destination or in a different bus. If the target destination is not within the same bus as the gateway destination, you must also specify the targetBus parameter. You must either specify the targetDestination parameter or the targetService parameter. (Conditional) **targetService** The name of the target outbound service. You must either specify the targetDestination parameter or the targetService parameter. (Conditional)	**Batch mode example usage:** • Using Jacl: `set gwTarget [$AdminTask addWSGWTargetService $gwService {-name "AnotherTarget" -targetService "AnotherService"}]` • Using Jython: `gwTarget=AdminTask.addWSGWTargetService(gwService, '[-name AnotherTarget -targetService AnotherService]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask addWSGWTargetService {-interactive}` • Using Jython: `AdminTask.addWSGWTargetService('[-interactive]')`
				targetBus The name of the WPM bus that contains the target. (Optional) • Returns: The object name of the target service object that you created.	

compare Node Version	Managed Object Metadata group	The **compare Node Version** command compares the WebSphere Application Server version given a node that you specify and an input version.	None	• Parameters: **- 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. • Returns: – 0 if node version matches the input version – -1 if node version is smaller than the input version – 1 is node version is higher than the input version	**Batch mode example usage:** • Using Jacl: `$AdminTask compare NodeVersion {-nodeName node1 -version 5}` • Using Jython: `AdminTask.compareNodeVersion('[-nodeName node1 -version 5]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask compare NodeVersion {-interactive}` • Using Jython: `AdminTask.compareNodeVersion ('[-interactive]')`
configure TAM					**Interactive mode example usage:** • Using Jacl: `$AdminTask configure TAM {-interactive}` • Using Jython: `AdminTask.configure TAM ('[-interactive]')`

connect SIBWS Endpoint Listener	SIBWeb Services group	The **connect SIBWS Endpoint Listener** command connects an end point listener to a bus.	Object name of the end point listener that you want to create.	• Parameters: **bus** The name of the bus to which the end point listener will be connected. (required) **replyDestination** The name of the reply destination for the connection. (optional) • Returns: The SIBWS bus connection property object.	**Batch mode example usage:** • Using Jacl: `set busConn [$Admin Task connectSIBWSEnd pointListener $epl {-bus "MyBus"}]` • Using Jython: `busConn = AdminTask. connectSIBWSEndpoint Listener(epl, '[-bus MyBus]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask connect SIBWSEndpointListener {-interactive}` • Using Jython: `AdminTask.connectSIB WSEndpointListener ('[-interactive]')`

copy Resource Adapter	JCA management group	Use the **copy Resource Adapter** command to create a Java 2 Connector (J2C) resource adapter under the scope that you specify.	J2C Resource Adapter_ object_ID	• Parameters: **- 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`. • Returns: J2C resource adapter object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask copy ResourceAdapter $ra [subst {-name newRA -scope $scope}]` • Using Jython: `AdminTask.copy ResourceAdapter(ra, '[-name newRA -scope scope]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask copy ResourceAdapter {-interactive}` • Using Jython: `AdminTask.copy ResourceAdapter ('[-interactive]')`
create Application Server					**Interactive mode example usage:** • Using Jacl: `$AdminTask create ApplicationServer {-interactive}` • Using Jython: `AdminTask.create ApplicationServer ('[-interactive]')`

create Application Server Template					**Interactive mode example usage:** • Using Jacl: `$AdminTask create ApplicationServer Template {-interactive}` • Using Jython: `AdminTask.create ApplicationServer Template ('[-interactive]')`

create Chain	Channel Framework Management group	The **create Chain** command creates a new chain of transport channels based on a chain template.	The instance of the transport channel service under which the new chain is created. (ObjectName, required)	• Parameters: **- 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) • Returns: The object name of the channel chain that was created.	**Batch mode example usage:** • Using Jacl: `$AdminTask create Chain (cells/ rohitbuildCell01/ nodes/rohitbuild CellManager01/ servers/dmgr\| server.xml# TransportChannel Service_1) {-template` *`WebContainer`* `(templates/chains\| webcontainer-chains. xml#Chain_1) -name` *`trialChain1`*`}` `$AdminTask create Chain (cells/ rohitbuildCell01/ nodes/rohitbuild CellManager01/ servers/dmgr| server.xml#Transport ChannelService_1) {-template` *`WebContainer`* `(templates/chains| webcontainer-chains. xml#Chain_1) -name` *`trialChain1`* `-endPoint (cells/ rohitbuildCell01/ nodes/rohitbuild CellManager01|server index.xml#EndPoint_3)}` • Using Jython: `AdminTask.create Chain('cells/rohit buildCell01/nodes/ rohitbuildCellManager 01/servers/dmgr| server.xml# TransportChannel Service_1', '[-temp late "WebContainer (templates/chains| webcontainer-chains. xml#Chain_1)" -name trialChain]')`

					AdminTask.createChain ('cells/rohitbuild Cell01/nodes/rohit buildCellManager01/ servers/dmgr|server. xml#TransportChannel Service_1', '[-template "WebContainer (templates/chains|web container-chains. xml#Chain_1)" -name trialChain -endPoint "(cells/rohitbuild Cell01/nodes/rohit buildCellManager01| serverindex.xml# EndPoint_3)"]') **Interactive mode example usage:** • Using Jacl: `$AdminTask createChain {-interactive}` • Using Jython: `AdminTask.createChain ('[-interactive]')`

create Cluster	Cluster Config Comm ands	The **create Cluster** command creates a new server cluster. A server cluster consists of a group of application servers which 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.	None	• **Parameters for step one:** **-clusterConfig** Specifies the configuration of the new server cluster. This command step is required. The following parameters can be specified for this step. **clusterName** The name of the new server cluster. This parameter is required. **preferLocal** Enables or disables node scoped routing optimization within this cluster. This parameter is optional. The value is true or false. It not specified, the default value is true. • **Parameters for step two:**	**Batch mode example usage:** • Using Jacl: `$AdminTask create Cluster { -cluster Config {{cluster1 true}}}` `$AdminTask create Cluster { -cluster Config {{cluster1 true}} -replication Domain {{true}}}` `$AdminTask create Cluster { -cluster Config {{cluster1 true}} -convert Server {{server1 node1 "" "" ""}}}` • Using Jython: `AdminTask.create Cluster('[-cluster Config [[cluster1 true]]]')` `AdminTask.create Cluster('[-cluster Config [[cluster1 true]] -replication Domain [[true]]]')` `AdminTask.create Cluster('[-cluster Config [[cluster1 true]] -convert Server [[server1 node1 "" "" ""]]]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask create Cluster {-interactive}` • Using Jython: `AdminTask.create Cluster ('[-inter active]')`

				-replicationDomain Specifies the configuration of a replication domain for this cluster. A replication domain is used to support HTTP session data replication. This command step is optional. The following parameters can be specified for this step: **createDomain** Creates a replication domain with a name set to the name of the new cluster. This parameter is optional. The value is true or false. It not specified, the default value is false. • **Parameters for step three:** **-convertServer** 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. This parameter is required for the command step. You must also specify the serverName parameter. **serverName** The name of the application server to be converted to the first cluster member. This parameter is required for the command step. You must also specify the serverNode parameter. **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.	

				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. This parameter is optional. 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. **replicatorEntry** Specifies a replicator entry for the converted member will be created in the cluster's 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 replicator entry will be created. The default value is false. You can specify this parameter only if the createDomain parameter was set to true in the replicationDomain command step. • **Returns:** ObjectName of cluster created.	

create Cluster Member	Cluster Config Commands	The **create Cluster Member** 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	cluster ObjectID - The configuration object ID of the cluster which the new member will belong to. If this is not specified, then the clusterName parameter must be specified. The object name can be obtained programmatically via Java using the WebSphere ConfigService API, or via wsadmin scripting using the AdminConfig command.	• **Parameters:** **-clusterName** The name of the cluster which the new member will belong to. If this parameter is not specified, then the cluster object ID must be specified in the command target. • **Parameters for step one:** **-memberConfig** Specifies the attributes of the new cluster member to be created in the cluster. This command step is required. The following parameters can be specified for this step: **memberName** The name of the server to be created for the new cluster member. This parameter is required.	**Batch mode example usage:** • Using Jacl: First member creation using template name: `$AdminTask create ClusterMember {-clusterName cluster1 -memberConfig {{node1 member1 "" "" true false}} -firstmember {{ serverTemplateName "" "" "" ""}}}` First member creation using server and node for template: `$AdminTask create ClusterMember {-clusterName cluster1 -memberConfig {{node1 member1 "" "" true false}} -firstmember {{ "" node1 server1 "" ""}}}` Second member creation: `$AdminTask create ClusterMember {-clusterName cluster1 -memberConfig {{node1 member2 "" "" true false}}}` • Using Jython: First member creation using template name: `AdminTask.create ClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member1 "" "" true false]] -firstMember [[server TemplateName "" "" "" ""]]]')` First member creation using server and node for template: `AdminTask.create ClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member1 "" "" true false]] -firstMember [["" node1 server1 "" ""]]]')` Second member creation: `AdminTask.create ClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member2 "" "" true false]]]')`

				memberNode The name of the node where the new cluster member will be created. This parameter is required. **memberWeight** The weight of the new cluster member. This 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. The default value is 2. **genUniquePorts** 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 value is true or false. The default value is true .	**Interactive mode example usage:** • Using Jacl: `$AdminTask create ClusterMember {-interactive}` • Using Jython: `AdminTask.create ClusterMember ('[-interactive]')`

				replicatorEntry Specifies a replicator entry for the new cluster member will be created in the cluster's 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. • **Parameters for step two:** **-firstMember** Specifies additional information necessary to create the first cluster member. This command step is required when creating the first member of the cluster, and is executable only when creating the first member of the cluster. The target of this command step is a Boolean value indicating whether or not to perform this step.	

				The default value is true if any of the step parameters are specified; otherwise the default value is false. The following parameters can be specified for this step: **templateName** 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. **templateServerNode** 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.	

				templateServerName 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. **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. This parameter is optional. 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.	

				coreGroup The name of the core group this cluster member, and all future cluster members, must belong to. All cluster members must belong to the same core group. This parameter is optional. If not specified, the default value is the default core group defined in the cell. • **Returns:** ObjectName of cluster member created.	
create Core Group	Core Group Management group	The **create Core Group** command creates a new core group. The core group that you create will contain no members.	None	• Parameters: **- coreGroupName** The name of the core group that you are creating. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createCoreGroup {-coreGroupName MyCoreGroup}` • Using Jython: `AdminTask.createCoreGroup('[-coreGroupName MyCoreGroup ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCoreGroup {-interactive}` • Using Jython: `AdminTask.createCoreGroup ('[-interactive ]')`

create Core Group Access Point	Core Group BridgeManagement group	The **create Core Group Access Point** 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.	Core group bridge settings object for the cell. (ObjectName, required).	• Parameters: **- coreGroupName** The name of the core group for which the core group access point will be created. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createCore GroupAccessPoint (cells/rohitbuild Cell01\|coregroup bridge.xml#CoreGroup BridgeSettings_1) "-coreGroupName DefaultCoreGroup"` • Using Jython: `AdminTask.createCore GroupAccessPoint ('cells/rohitbuild Cell01\|coregroup bridge.xml#CoreGroup BridgeSettings_1', '[-coreGroupName DefaultCoreGroup]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCore GroupAccessPoint {-interactive}` • Using Jython: `AdminTask.createCore GroupAccessPoint ('[-interactive]')`
create Default CGAP					**Interactive mode example usage:** • Using Jacl: `$AdminTask create DefaultCGAP {-interactive}` • Using Jython: `AdminTask.create DefaultCGAP ('[-interactive]')`

create Generic Server	Server Management group Step: Config ProcDef	Use the **create Generic Server** 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 **create Generic Server** command provides an additional step, ConfigProcDef, that you can use to configure the parameters that are specific to generic servers.	None	• Parameters: **- name** The name of the server that you want to create. **- 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. **- templateLocation** The location of the server template. **- startCommand** Indicates the path to the command that will run when this generic server is started. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask create GenericServer jim667BaseNode {-name jgeneric -ConfigProc Def {{"/usr/bin/my StartCommand" "arg1 arg2" "" "" "/tmp/ workingDirectory" "/tmp/stopCommand" "argy argz"}}}` • Using Jython: `AdminTask.create GenericServer(jim667 BaseNode, '[-name jgeneric -ConfigProc Def [[/usr/bin/myStart Command "arg1 arg2" "" "" /tmp/working Directory /tmp/Stop Command "argy argz "]]]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask create GenericServer {-interactive}` • Using Jython: `AdminTask.create GenericServer ('[-interactive]')`

				- 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) • Returns: null	
create Generic Server Template					**Interactive mode example usage:** • Using Jacl: `$AdminTask create GenericServerTemplate {-interactive}` • Using Jython: `AdminTask.create GenericServerTemplate ('[-interactive]')`

create J2CActivationSpec	JCA management group	Use the **create J2C Activation Spec** command to create a Java 2 Connector (J2C) activation specification under a J2C resource adapter and attributes that you specify. Use the messageListenerType parameter to indicate the activation specification that is defined for the J2C resource adapter.	J2C Resource Adapter _object_ID	• Parameters: **- 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.	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2C ActivationSpec $ra {-name J2CActSpec -jndiName eis/ActSpec1 -messageListenerType javax.jms.Message Listener }` • Using Jython: `AdminTask.createJ2C ActivationSpec(ra, '[-name J2CActSpec -jndiName eis/ActSpec1 -messageListenerType javax.jms.Message Listener]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2C ActivationSpec {-interactive}` • Using Jython: `AdminTask.createJ2C ActivationSpec ('[-interactive]')`
				- authenticationAlias Indicates the authentication alias of the J2C activation specification that you are creating. **- description** Description of the created J2C activation spec. • Returns: J2CActivationSpec object ID	

create J2C Admin Object	JCA management group	Use the **create J2C Admin Object** command to create a administrative object under a resource adapter with attributes that you specify. Use the administrative object interface to indicate the administrative object defined in the resource adapter.	J2CResourceAdapter_object_ID	• Parameters: **-adminObject Interface** 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. • Returns: J2CAdminObject object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2C AdminObject $ra {-adminObjectInterface fvt.adapter.message.FVTMessageProvider -name J2CA01 -jndiName eis/J2CA01}` • Using Jython: `AdminTask.createJ2C AdminObject(ra, '[-adminObjectInterface fvt.adapter.message.FVTMessageProvider -name J2CA01 -jndiName eis/J2CA01]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2C AdminObject {-interactive}` • Using Jython: `AdminTask.createJ2C AdminObject ('[-interactive]')`

createJ2C Conne ction Factory	JCA manag ement group	Use the **create J2C Conne ction Factory** command to create a Java 2 connection factory under a Java 2 resource adapter and attributes that you specify. Use the connection factory interfaces to indicate the connection definitions defined for the Java 2 resource adapter.	J2C Connection Factory_ object_ID	• Parameters: **-connectionFactory Interface** 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. • Returns: J2C connectionfactory object ID.	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2C ConnectionFactory $ra {-connectionFactory Interfaces javax. sql.DataSource -name J2CCF1 -jndiName eis/J2CCF1}` • Using Jython: `AdminTask.createJ2C ConnectionFactory (ra, '[-connection FactoryInterfaces javax.sql.Data Source -name J2CCF1 -jndiName eis/J2CCF1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2C ConnectionFactory {-interactive}` • Using Jython: `AdminTask.createJ2C ConnectionFactory ('[-interactive]')`
create Node Group	Node Group Comma nds group	The **create Node Group** command creates a new node group. A node group consists of a group of nodes which are referred to as node group members. Optionally, you can create a short name and description for the new node group.	The node group name to be created. This target object is required.	• Parameters: **- shortName** The short name of the node group. This parameter is optional. **- description** The description of the node group. This parameter is optional. • Returns: Node group object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask createNode Group WBINodeGroup` • Using Jython: `AdminTask.createNode Group('WBINodeGroup')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createNode Group {-interactive}` • Using Jython: `AdminTask.createNode Group ('[-interactive]')`

create Node Group Property	Node Group Commands group	The **create Node Group Property** command creates custom properties for a node group.	The name of the node group. This target object is required.	• Parameters: **- 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. • Returns: Properties object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask createNodeGroupProperty WBINodeGroup {-name Channel -value "channel1"}` • Using Jython: `AdminTask.createNodeGroupProperty('WBINodeGroup', '[-name Channel -value channel1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createNodeGroupProperty {-interactive}` • Using Jython: `AdminTask.createNodeGroupProperty ('[-interactive]')`

create SIB Destin ation	SIB Admin Comm ands	Use this command to create a SIB destination.	None	• Parameters: **bus** name of the bus where this destination is to be configured (String, required) **name** destination name (String, required) **type** The destination type. Valid value include: Queue, TopicSpace, WebService or Port. If the type is not TopicSpace, you must use the node/server or cluster option to specify a bus member. (String, required) **cluster** to assign the destination to a cluster, supply cluster name, but not node and server name. (optional) **node** to assign the destination to a server, supply node name server name, but not cluster name. (optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB Destination {-bus` *`busname`* `-name` *`destname`* `-type TopicSpace}` • Using Jython: `AdminTask.createSIB Destination('[-bus` *`busname`* `-name` *`destname`* `-type TopicSpace]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB Destination {-interactive}` • Using Jython: `AdminTask.createSIB Destination ('[-interactive]')`

				server to assign the destination to a server, supply node name server name, but not cluster name. (optional) **aliasBus** if this is an alias destination, the source bus name of alias mapping. (optional) **targetBus** if this is an alias destination, the name of the bus that the destination it maps to is configured on. (optional) **targetName** if this is an alias destination, the name of the destination it maps to. (optional) **foreignBus** if this is a foreign destination, the name of the foreign bus. (optional)	

				description description. (optional) **reliability** the reliability quality of service for message flows through this destination, from BEST_EFFORT_NON-PERSISTENT to ASSURED_PERSISTENT, in order of increasing reliability. Higher levels of reliability have higher impacts on the performance. (optional) **maxReliability** the maximum reliability quality of service that is accepted for values specified by producers. (optional) **overrideOfQOSByProducerAllowed** controls the quality of service for message flows between producers and the destination. Select this option to use the quality of service specified by producers instead of the quality defined for the destination. (optional)	

				defaultPriority the default priority for message flows through this destination, in the range 0 (lowest) through 9 (highest). This default priority is used for messages that do not contain a priority value (Integer, optional). (optional) **maxFailedDeliveries** the maximum number of times that service tries to deliver a message to the destination before forwarding it to the exception destination (Integer, optional). (optional) **exceptionDestination** the name of another destination to which the system sends a message that cannot be delivered to this destination within the specified maximum number of failed deliveries. (optional)	

				sendAllowed clear this option (setting it to false) to stop producers from being able to send messages to this destination. (optional) **receiveAllowed** clear this option (setting it to false) to prevent consumers from being able to receive messages from this destination. (optional) **quiesceMode** select this option (setting it to true) to indicate that the destination is quiescing. In quiesce mode, new messages for the destination cannot be added to the bus, but any messages already in the bus can still be sent to, and processed by, the destination (Boolean, optional, default=False). (optional) **receiveExclusive** select this option (setting it to true) to allow only one consumer to attach to a destination (Boolean, optional, default=False). (optional)	

				topicAccessCheckRequired topic access check required (Boolean, optional) **replyDestination** clear this option (setting it to false) to stop producers from being able to send messages to this destination. (optional) **replyDestinationBus** clear this option (setting it to false) to prevent consumers from being able to receive messages from this destination. (optional) **delegateAuthorizationCheckToTarget** indicates whether the authorization check should be delegated to the alias or target destination (Boolean, optional)	
				• **Parameters for step one:** **defaultForwardRoutingPath** the default forward routing path. **bus** bus name **destination** destination name • **Returns:** A new SIB destination.	

create SIB Engine	SIBAd min Comm ands	Use the **create SIB Engine** command to create a new messaging engine for a bus member.	None	• Parameters: **bus** name of the bus to which the messaging engine is to belong (String, optional) **node** to create a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **server** to create a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **cluster** to create a messaging engine on a cluster, supply cluster name, but not node and server name (String, optional) **description** description of the messaging engine (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB Engine {-bus busname -node nodeName -server severname}` • Using Jython: `AdminTask.createSIB Engine('[-bus busname -node nodeName -server severname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB Engine {-interactive}` • Using Jython: `AdminTask.createSIB Engine ('[-interactive]')`

				initialState Indicates if the messaging engine is started or stopped when the associated application server starts. Until started, the messaging engine is unavailable. Valid values are `Stopped` and `Started`. (String, optional) **destinationHighMsgs** the maximum total number of messages that the messaging engine can place on its message points (Long, optional) **createDefaultDatasource** Set to `true` if a default data source should be created when the messaging engine is created (Boolean, optional) **datasourceJndiName** JNDI name of the data source to be referenced from the datastore created when the messaging engine is created (String, optional) • Returns: A new SIB messaging engine.	

create SIB JMS Activation Spec	SIB JMS Admin Commands	Use the **create SIB JMS Activation Spec** command to create a SIB JMS activation specification.	Scope of the SIB JMS resource adapter to which the activation specification will be added.	• Parameters: **name** name of new activation specification (String, required) **jndiName** JNDI name of the activation specification (String, required) **destinationJndiName** JNDI name of a destination (String, required) **description** a JMS activation specification is used by the default messaging provider to validate the activation-configuration properties for a JMS message-driven bean (MDB) (String, optional) **acknowledgeMode** how the session acknowledges any messages it receives (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB JMSActivationSpec $ra {-name specname -jndiName specname}` • Using Jython: `AdminTask.createSIB JMSActivationSpec (ra, '[-name specname -jndiName specname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB JMSActivationSpec {-interactive}` • Using Jython: `AdminTask.createSIB JMSActivationSpec ('[-interactive]')`

				authenticationAlias authentication alias (String, optional) **busName** name of the SIB bus to connect to (String, required) **clientId** client identifier. Required for durable topic subscriptions (String, optional) **destinationType** Indicates if the message-driven bean uses a queue or topic destination. (String, optional) **durableSubscriptionHome** The name of the durable subscription home. This identifies the messaging engine where all durable subscriptions accessed through this activation specification are managed (String, optional)	

				maxBatchSize the maximum number of messages received from the messaging engine in a single batch (Integer, optional) **maxConcurrency** the maximum number of endpoints to which messages are delivered concurrently (Integer, optional) **messageSelector** the JMS message selector used to determine which messages the message-driven bean (MDB) receives (String, optional) **password** password (String, optional) **subscriptionDurability** whether a JMS topic subscription is durable or nondurable (String, optional)	
				subscriptionName the subscription name needed for durable topic subscriptions (String, optional) **shareDurableSubscriptions** used to control how durable subscriptions are shared (String, optional, default = AsCluster) **userName** user name (String, optional) • Returns: A new SIB JMS activation specification.	

create SIB JMS Connection Factory	SIB JMS Admin Commands	Use the **create SIB JMS Connection Factory** command to create a generic, queue or topic SIB JMS connection factory.	Scope of the SIB JMS resource adapter to which the SIB JMS connection factory will be added.	• Parameters: **name** The name of the SIB JMS connection factory (String, required) **jndiName** the JNDI name of the SIB JMS connection factory (String, required) **type** The type of connection factory to create. To create a queue connection factory, set the value to `Queue`. To create a topic connection factory, set to `Topic`. To create a generic connection factory, do not set a value. (String, optional) **authDataAlias** Specifies a user ID and password to be used to authenticate connections to the JMS provider for application-managed authentication (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB JMSConnectionFactory $ra {-name connectionfactory_name -jndiName jndi_name}` • Using Jython: `AdminTask.createSIB JMSConnectionFactory (ra, '[-name connectionfactory_name -jndiName jndi_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB JMSConnectionFactory {-interactive}` • Using Jython: `AdminTask.createSIB JMSConnectionFactory ('[-interactive]')`

				category classifies or groups the connection factory (String, optional) **description** description of the connection factory (String, optional) **logMissingTransactionContext** whether or not the container logs that there is a missing transaction context when a connection is obtained (Boolean, optional, default = False) **manageCachedHandles** Indicates if cached handles (handles held in instance variables in a bean) should be tracked by the container (Boolean, optional, default = False)	
				xaRecoveryAuthAlias the authentication alias used during XA recovery processing (String, optional) **busName** the SIB bus name (String, optional) **clientID** user-defined string, only required for durable subscriptions (String, optional) **userName** The user name that is used to create connections from the connection factory (String, optional) **password** the password that is used to create connections from the connection factory (String, optional)	

				nonPersistentMapping non-persistent mapping value. Valid values include: `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination` and `None` (String, optional) **persistentMapping** persistent mapping value. Valid values include: `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination` and `None` (String, optional)	
				durableSubscriptionHome durable subscription home value (String, optional) **readAhead** read-ahead value. Valid values include: `Default`, `AlwaysOn` and `AlwaysOff` (String, optional) **target** the name of a target that resolves to a group of messaging engines (String, optional) **targetType** specifies the type of the name in the target parameter. Valid values are `BusMember`, `Custom` and `ME` (String, optional) **targetSignificance** this property specifies the significance of the target group (String, optional)	

				remoteProtocol the name of the protocol that should be used to connect to a remote messaging engine (String, optional) **providerEndPoints** A list of endpoint triplets seperated by commas, for example: `host:port:chain` (String, optional) **connectionProximity** the proximity of acceptable messaging engines. Valid values include: `Bus`, `Host`, `Cluster` and `Server` (String, optional) **tempQueueNamePrefix** temporary queue name prefix (String, optional) **tempTopicNamePrefix** temporary topic name prefix (String, optional)	
				shareDataSourceWithCMP used to control how data sources are shared (Boolean, optional) **shareDurableSubscriptions** used to control how durable subscriptions are shared. Legal values are ″AsCluster″, ″AlwaysShared″ and ″NeverShared″ (String, optional, default = AsCluster) • Returns: A new SIB JMS connection factory.	

create SIB JMS Queue	SIB JMS AdminCommands	Use the **create SIB JMS Queue** command to create a SIB JMS queue.	Scope of the SIB JMS resource adapter to which the SIB JMS queue will be added.	• Parameters: **name** The name of the SIB JMS queue. (String, required) **jndiName** The JNDI name of the SIB JMS queue. (String, required) **description** A description of the SIB JMS queue (String, optional) **queueName** The name of the underlying SIB queue to which the queue points (String, required) **deliveryMode** The delivery mode for messages. Legal values are ″Application″, ″NonPersistent″ and ″Persistent″ (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB JMSQueue $ra {-name queue_name -jndiName jndi_name -queueName queue_name}` • Using Jython: `AdminTask.createSIB JMSQueue(ra, '[-name queue_name -jndiName jndi_name -queueName queue_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB JMSQueue {-interactive}` • Using Jython: `AdminTask.createSIB JMSQueue ('[-interactive]')`

				timeToLive the time in milliseconds to be used for message expiration (Long, optional) **priority** the priority for messages. Whole number in the range 0 to 9 (Integer, optional) **readAhead** read-ahead value. Legal values are ″AsConnection″, ″AlwaysOn″ and ″AlwaysOff″ (String, optional) **busName** the name of the bus on which the queue resides (String, optional) • Returns: A new SIB JMS queue.	

create SIB JMS Topic	SIB JMS Admin Comm ands	Use this command to create a SIB JMS topic.	Scope of the SIB JMS resource adapter to which the SIB JMS topic will be added.	• Parameters: **name** The name of the SIB JMS topic (String, required) **jndiName** the SIB JMS topic's JNDI name (String, required) **description** a description of the SIB JMS queue (String, optional) **topicSpace** the name of the underlying SIB topic space to which the topic points (String, required) ***topicName** the topic to be used inside the topic space (for example, stock/IBM) (String, required)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB JMSTopic $ra {-name topic_name -jndiName jndi_name -topicName topic_name -topicSpace topicspace_name}` • Using Jython: `AdminTask.createSIB JMSTopic(ra, '[-name topic_name -jndiName jndi_name -topicName topic_name -topicSpace topic space_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIBJ MSTopic {-interactive}` • Using Jython: `AdminTask.createSIBJ MSTopic ('[-interactive]')`

				deliveryMode the delivery mode for messages. Legal values are ″Application″, ″NonPersistent″ and ″Persistent″ (String, optional) **timeToLive** the time in milliseconds to be used for message expiration (Long, optional) **priority** the priority for messages. Whole number in the range 0 to 9 (Integer, optional) **readAhead** read-ahead value. Legal values are ″AsConnection″, ″AlwaysOn″ and ″AlwaysOff″ (String, optional) **busName** the name of the bus on which the topic resides (String, optional) • Returns: A new SIB JMS topic.	

create SIB Mediation	SIB Admin Commands	Use this command to create a SIB mediation.	None	• Parameters: **bus** name of the bus where the mediation is to be created (String, required) **mediationName** name to be given to the mediation (String, required) **description** description of the mediation (String, optional) **handlerListName** name of the handler list that was defined when the mediation was deployed (String, required) **globalTransaction** whether or not a global transaction is started for each message processed (Boolean, optional, default = False)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIB Mediation {-bus` *bus_name* `-mediationName` *mediation_name* `-handlerListName` *handlerList_name*`}` • Using Jython: `AdminTask.createSIB Mediation('[-bus` *bus_name* `-mediationName` *mediation_name* `-handlerListName` *handlerList_name* `]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIB Mediation {-interactive}` • Using Jython: `AdminTask.createSIB Mediation ('[-interactive]')`

				allowConcurrentMediation whether or not to apply the mediation to multiple messages concurrently, and preserve message ordering (Boolean, optional, default = False) **selector** the text string that must be present in a message for the mediation to process the message (String, optional) **discriminator** the text string that must not be present in a message for the mediation to process the message (String, optional) • Returns: A new SIB mediation.	

create SIBWS Endpoint Listener	SIB Web Services group	The **create SIB WS Endpoint Listener** command creates an end point listener object with no SIBWS bus connection property objects.	Object name of the server where the end point listener will be created.	• Parameters: **name** The name of the end point listener within the server. (required) **urlRoot** The root of the end point address URL for Web services that you access through the end point listener. (required) **wsdlUrlRoot** The root of the HTTP URL where you can retrieve the WSDL associated with this end point listener. (required) • Returns: The SIBWS end point object.	**Batch mode example usage:** • Using Jacl: `set epl [$AdminTask createSIBWSEndpointListener $server {-name "soaphttp1" -urlRoot "http://myserver.com/wsgwsoaphttp1" -wsdlUrlRoot "http://myserver.com/wsgwsoaphttp1"}]` • Using Jython: `epl = AdminTask.createSIBWSEndpointListener (server, '[-name soaphttp1 -urlRoot http://myserver.com/wsgwsoaphttp1 -wsdlUrlRoot http://myserver.com/wsgwsoaphttp1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIBWSEndpointListener {-interactive}` • Using Jython: `AdminTask.createSIBWSEndpointListener ('[-interactive]')`

create SIBWS Inbound Service	SIB Web Services group	The **create SIBWS Inbound Service** command creates a new inbound service object that represents a protocol attachment that service requesters will use. If you specify the UDDIReference option, the wsdlLocation option is assumed to be a UDDI service key in the following format where each n is a hex digit: nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn.	The object name of the messaging bus within which the service will be created.	• Parameters: **name** The administrative name of the inbound service. (required) **destination** The name of the underlying WPM destination. (required) **wsdlLocation** The location of the template WSDL. The value of this parameter can be a URL or a UDDI service key (UUID). (required) **wsdlServiceName** The name of the service in the WSDL. You must specify this parameter or the wsdlServiceNamespace parameter. (conditional) **wsdlServiceNamespace** The namespace of the service in the WSDL. You must specify this parameter or the wsdlServiceName parameter. (conditional)	**Batch mode example usage:** • Using Jacl: `set inService [$AdminTask createSIBWSInboundService $bus {-name "MyService" -destination $destName -wsdlLocation "http://myserver.com/MyService.wsdl"}]` • Using Jython: `inService = AdminTask.createSIBWSInboundService(bus, '[-name MyService -destination destName -wsdlLocation http://myserver.com/MyService.wsdl]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIBWSInboundService {-interactive}` • Using Jython: `AdminTask.createSIBWSInboundService ('[-interactive]')`

				uddiReference The reference of the UDDI registry for the WSDL. (optional) **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: The object name of the created inbound service object.	
create SIBWS Outbound Service	SIBWeb Services group	The **create SIBWS Outbound Service** command creates a new outbound service object that represents a protocol attachment to a service provider. This command requires the identification of a single service element within a WSDL document.	The object name of the messaging bus within which the service is created.	• Parameters: **name** The administrative name of the outbound service. (required) **wsdlLocation** The location of the WSDL of the service provider. It can be a URL or a UDDI service key (UUID). (required) **wsdlServiceName** The name of the service in the WSDL. You must specify the wsdlServiceName parameter or the wsdlServiceNamespace parameter. (conditional) **wsdlServiceNamespace** The namespace of the service in the WSDL. You must specify the wsdlServiceName parameter or the wsdlServiceNamespace parameter. (conditional)	**Batch mode example usage:** • Using Jacl: `set outService [$Admin Task createSIBWSOut boundService $bus {-name "MyServi ce" -wsdlLocation "http://myserver.co m/MyService.wsdl"}]` • Using Jython: `outService = Admin Task.createSIBWSOut boundService(bus, '[-name MyServi ce -wsdlLocation http://myserver.co m/MyService.wsdl]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIBW SOutboundService {-interactive}` • Using Jython: `AdminTask.createSIBW SOutboundService ('[-interactive]')`

				uddiReference The reference of the UDDI registry for the WSDL. (optional) **destination** The name of the service destination. (optional) **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: The object name of the outbound service object that was created.	

create SIBus	SIB Admin Commands	Use this command to create a new bus on the current node.	None	• Parameters: **bus** name of bus to create, which must be unique in the cell (String, required) **description** descriptive information about the bus (String, required) **secure** enable or disable bus security (Boolean, optional, default = False) **interEngineAuthAlias** name of the authentication alias used to authorize communication between messaging engines on the bus (String, optional) **mediationsAuthAlias** name of the authentication alias used to authorize mediations to access the bus (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSIBus {-bus busname -description text -secure True -mediationsAuthAlias name -protocol protocol -discardOnDelete False}` • Using Jython: `AdminTask.createSIBus ('[-bus busname -description "text" -secure True -mediationsAuthAlias name -protocol protocol -discardOnDelete False]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSIBus {-interactive}` • Using Jython: `AdminTask.createSIBus ('[-interactive]')`

				protocol the protocol used to send and receive messages between messaging engines, and between API clients and messaging engines (String, optional) **discardOnDelete** indicate whether or not any messages left in the data store of a queue should be discarded when the queue is deleted (Boolean, optional, default = False) **destinationHighMsgs** the maximum number of messages that any queue on the bus can hold (Long, optional) **configurationReloadEnabled** indicate whether configuration files should be dynamically reloaded for this bus (Boolean, optional, default = True) • Returns: A new SIB bus.	

create Server Type	None	Use the **create Server Type** command to define a server type.	None	• Parameters: **-version** (String, required) **-serverType** (String, required) **-createTemplate Command** (String, required) **-createCommand** (String, required) **-defaultTemplateName** The default value is: default. (String, optional) **-configValidator** (String, optional) • Returns: The identification of the server type that you created, `javax.management. ObjectName` .	**Batch mode example usage:** • Using Jacl: `$AdminTask createSer verType {-version` *`version`* `-server Type` *`serverType`* `-createTemplateCom mand` *`name`* `-createCommand` *`name`*`}` • Using Jython: `AdminTask.createServ erType('[-version` *`version`* `-server Type` *`serverType`* `-createTemplateCo mmand` *`name`* `-createCommand` *`name`*`]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSer verType {-interactive}` • Using Jython: `AdminTask.createSer verType ('[-inter active]')`

createTCP EndPoint	None	The **createTCP EndPoint** command creates a new named end point you can associate with a TCP inbound channel.	Parent instance of the Transport Channel Service that contains the TCP Inbound Channel. (ObjectName, required)	• Parameters: **-name** Name for the new NamedEndPoint. (String, required) **- host** Host for the new NamedEndPoint. (String, required) **- port** Port for the new NamedEndPoint. (String, required) • Returns: Object name of the created NamedEndPoint.	**Batch mode example usage:** • Using Jacl: `$AdminTask createTCP EndPoint (cells/rohit buildCell01/nodes/ rohitbuildCellManag er01/servers/dmgr\| server.xml#Transpor tChannelService_1) {-name Sample_End_ Pt_Name -host rohitbuild.ralei gh.ibm.com -port 8978}` • Using Jython: `AdminTask.createTCPEnd Point('cells/rohitbui ldCell01/nodes/rohit buildCellManager01/ servers/dmgr\|server. xml#TransportChannel Service_1', '[-name Sample_End_Pt_Name -host rohitbuild. raleigh.ibm.com -port 8978]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createTCPE ndPoint {-interactive}` • Using Jython: `AdminTask.createTCPE ndPoint ('[-interac tive]')`

create Unmanaged Node	Unmanaged Node Commands group	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 nor a deployment manager. Unmanaged nodes may contain Web servers, such as IBM IHS server.	None	• Parameters: **- 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) • Returns: null	**Batch mode example usage:** • Using Jacl: `$AdminTask createUnmanagedNode {-nodeName myNode-hostName myHost -nodeOperatingSystem linux}` • Using Jython: `AdminTask.createUnmanagedNode('[-nodeName jjNode -hostName jjHost -nodeOperating System linux]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createUnmanagedNode {-interactive}` • Using Jython: `AdminTask.createUnmanagedNode ('[-interactive]')`

create WSGW Gateway Service	WS Gateway group	The **create WSGW Gateway Service** command creates a new Gateway Service with associated InboundService and TargetService object. Configuration of the InboundPort and Outbound Service/Port associated with these is done using separate commands.	ObjectName of the gateway instance which the gateway service is created	• Parameters: **-name** Administrative name of the Gateway Service. (required) **-wsdlLocation** Location of the template WSDL. May be a URL or a UDDI business key (UUID). (conditional) **-wsdlServiceName** The name of the service in the WSDL. (conditional) **-wsdlServiceNamespace** The namespace of the service in the WSDL. (conditional) **-targetDestination** The name of the target destination. (conditional) **-targetService** The name of the target outbound service. (conditional)	**Batch mode example usage:** • Using Jacl: `set gwService [$Admin Task createWSGWGatew ayService $wsgw {-name MyGateway Service -targetSer vice MyService}]` • Using Jython: `gwService = AdminTask .createWSGWGateway Service(wsgw, '[-name MyGatewayService -targetService MyService]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSGW GatewayService {-interactive}` • Using Jython: `$AdminTask createWSGW GatewayService ('[-interactive]')`

				-requestDestination The name of the gateway destination. (optional) **-replyDestination** The name of the gateway reply destination. (optional) **-targetBus** The name of the WPM bus containing the target. (optional) **-uddiReference** The reference of the UDDI registry for the WSDL. (optional) **-userId** The user id to use to retrieve the WSDL. (optional) **-password** The password to use to retrieve the WSDL. (optional) • Returns: ObjectName of the created GatewayService object	

create WSGW Proxy Service	WS Gateway group	The **create WSGW Proxy Service** command creates a new proxy service with an associated inbound service, and a target service object with an associated outbound service. Configuration of the inbound port objects associated with the inbound service is done using separate commands.	The object name of the gateway instance within which the proxy service is created.	• Parameters: **name** The administrative name of the proxy service. (required) **node** The node where the destinations will be localized. (conditional) **server** The server where the destinations will be localized. (conditional) **cluster** Cluster where the destinations will be localized. (conditional) **-requestDestination** The name of the proxy request destination. (optional) **-replyDestination** The name of the proxy reply destination. (optional) **-wsdlLocation** The location of the proxy WSDL (URL). (optional) • Returns: The object name of the proxy service object that you created.	**Batch mode example usage:** • Using Jacl: `set proxyService [$AdminTask createWSGWProxyService $wsgw {-name MyProxyService -node MyNode -server server1}]` • Using Jython: `proxyService = AdminTask.createWSGWProxyService(wsgw, '[-name MyProxyService -node MyNode -server server1 ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSGWProxyService {-interactive}` • Using Jython: `AdminTask.createWSGWProxyService ('[-interactive]')`

create WebServer	Server Management group	Use the **create Web Server** command to create a Web server definition. This command is a two step process. The first step creates a Web server definition using a template. The parameters of the second step configure the Web server definition properties. Web server definitions generate and propagate the `plugin-config.xml` file for each Web server. For IHS only, the Web server definitions allow you to administer and configure IHS Web servers using the administrative console.	None	• **Parameters for step one:** **nodeName** The name of the node. (String, required) **name** The name of the server. (String, required) **templateName** 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. (optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createWeb Server {-name` *`web1`* `-serverConfig {{` *`webPort WebserverInstall Root`* `PluginInsta llRoot` *`Configura tion_file_name Windows_Server_ Name errorLog Path accessLog Path WebProto col`*`}} -remoteServer Config {{`*`AdminPort UserID Password AdminProtocol`*`}}` • Using Jython: `AdminTask.createWebSe rver('[-name` *`web1`* `-serverConfig [[web Port` *`Webserver InstallRoot`* `Plugin InstallRoot` *`Con figuration_file_name`* `Windows_Server_Name errorLogPath access LogPath WebProtocol]] -remoteServerConfig [[AdminPort` *`UserID`* `Password` *`AdminProt ocol`*`]]]')` where `-serverConfig` is second step of the command. – WebPort - is the port for the Webserver (required for all webservers) – WebserverInstallRoot - is the install path (directory) for webserver. necessary for IHS Admin Function. – Plugin Install Root - is install root where the plugin for the webserver is installed. Necessary for all webservers.

		These functions include the following: Start, Stop, View logs, View and Edit configuration file.		• **Parameters for step two:** **serverConfig** Create the Web server. (String, required) **webPort** The port for the Web server. (String, required) **configurationFile** The configuration file. The default is the path relative to the installation root, for example, `conf/httpd.conf`. (String, optional) **webInstallRoot** The installation path for the Web server. (String, required) **pluginInstallRoot** The plug-in installation path. (String, required)	• Configuration file name - is the file path for the IBM HTTP Server. This is necessary for View and edit of the IHS Configuration file only. • Windows Service Name - The windows service name on which IHS is to be started. This is necessary for Start and stop of the IHS webserver only. • ErrorLogPath - This is the path for the IHS error log (error.log) • AccessLogPath - This is the path for the IHS access log (access.log) • WebServerProtocol - HTTP or HTTPS where -remoteServerConfig is 3rd step of the command These parameters are only necessary if the IHS webserver is installed on a machine remote from WebSphere. • Admin Server Port - This is the port for the ADministration server. The administration server is installed on the same machine as the IBM HTTP Server. The admin server handles admin request to the IHS webserver. • UserID - This is the userID for authentication, if authentication is activated on the Administration server in the admin configuration file (admin.conf). • Passwd - This is the password for the specified authentication UserID. The password is generated by htpasswd utility in the admin.passwd file. • Admin ServerProtocol - HTTP or HTTPS **Interactive mode example usage:** • Using Jacl: `$AdminTask createWeb Server -interactive` • Using Jython: `AdminTask.createWeb Server ('[-intera ctive]')`

				serviceName The service name. (String, optional) **errorLogfile** The error log for viewing. The default is the path relative to the installation root, for example, `logs/error_log`. (String, optional) **accessLogfile** The access log for viewing. The default is the path relative to the installation root, for example, `logs/access_log`. (String, optional) **webProtocol** Parameters for the IHS administration server running with an unmanaged or remote Web server. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)	
				adminPort The port of the IHS administrative server. (String, required) **adminUserID** The user ID. This value should match the one for authentication in the `admin.conf`. (String, required) **adminPasswrd** The administrative password. (String, required) **adminProtocol** The administrative protocol title. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)	

				• **Parameters for step three:** Parameters for IHS administration server running with an unmanaged or remote Web server (installed on machine different from WebSphere Application Server) **adminPortTitle (adminPort)** Port of IHS administration **adminUserIDTitle (adminUserID)** The user ID. This value should match the authentication in the admin.conf file. **adminPasswdTitle (adminPasswd)** password **AdminProtocolTitle (adminProtocol)** This parameter is required. The value is either HTTP or HTTPS. The default value is HTTP. • **Returns:** None	

delete Chain	Channel Framework Management group	The **delete Chain** command deletes an existing chain and, optionally, the transport channels in the chain.	The chain to be deleted. (ObjectName ,required)	• Parameters: **- deleteChannels** If the value of this attribute is true, non-shared transport channels used by the specified chain will be deleted. (Boolean, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteChain trialChain1(cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr\|server.xml#Chain_1093554462922)` `$AdminTask deleteChain trialChain(cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#Chain_1093554378078) {-deleteChannels true}` • Using Jython: `AdminTask.deleteChain('trialChain1(cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)')` `AdminTask.deleteChain ('trialChain1(cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)', '[-deleteChannels true]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteChain {-interactive}` • Using Jython: `AdminTask.deleteChain ('[-interactive]')`

delete Cluster	Cluster Config Commands	The delete Cluster command deletes the configuration of a server cluster. A server cluster consists of a group of application servers which are referred to as cluster members. When a server cluster is deleted, all of its members are deleted. Use the delete Cluster Member command to delete the configuration of an individual cluster member.	cluster ObjectID - The configuration object ID of the cluster to be deleted. If the cluster's object ID is not specified, then the cluster Name parameter must be specified. The object name can be obtained programmatically via Java using the WebSphere Config Service API, or via wsadmin scripting using the AdminConfig command.	• **Parameters:** **-clusterName** The name of the cluster to be deleted. If this parameter is not specified, then the cluster object ID must be specified in the command target. • **Parameters for step one:** **-replicationDomain** Specifies the removal of the replication domain for this cluster. This command step is optional. The following parameters can be specified for this step:	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCluster { -clusterName cluster1 }` `$AdminTask deleteCluster { -clusterName cluster1 -replicationDomain {{true}}}` • Using Jython: `AdminTask.deleteCluster('[-clusterName cluster1]')` `AdminTask.deleteCluster('[-clusterName cluster1 -replicationDomain [[true]]]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCluster -interactive` • Using Jython: `AdminTask.deleteCluster ('[-interactive]')`
				deleteDomain Deletes the replication domain for this cluster. This parameter is optional. The value is true or false which indicates whether the domain will be deleted. The default value is false. . Deleting the replication domain deletes all replicator entries defined in the domain. • **Returns:** None	

delete Cluster Member	Cluster Config Commands	The **delete Cluster Member** command deletes the configuration of a cluster member. A cluster member is an application server that belongs to a server cluster. Use the **delete Cluster** command to delete the configuration of a cluster.	member ObjectID - The configuration object ID of the cluster member to be deleted. If this is not specified, then the cluster Name, member Node and member Name parameters must be specified. The object name can be obtained programmatically via Java using the WebSphere Config Service API, or via wsadmin scripting using the Admin Config command.	• **Parameters:** **-clusterName** The name of the cluster which the member to be deleted belongs to. 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. **-memberName** The server name of the member to be deleted 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.	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteClusterMember {-clusterName cluster1 -memberNode node1 -memberName member1}` `$AdminTask deleteClusterMember {-clusterName cluster1 -memberNode node1 -memberName member2 -replicationEntry {{true}}}` • Using Jython: `AdminTask.deleteClusterMember('[-clusterName cluster1 -memberNode node1 -memberName member1]')` `AdminTask.deleteClusterMember('[-clusterName cluster1 -memberNode node1 -memberName member2 -replicationEntry [[true]]]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteClusterMember -interactive` • Using Jython: `AdminTask.deleteClusterMember('[-interactive]')`

				-memberNode The name of the node having the cluster member to be deleted. 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. • **Parameters for step one:** **-replicatorEntry** 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 this cluster member's name from the cluster's replication domain. This parameter is optional. The value is true or false which indicates whether to delete the replicator entry. The default value is false. • **Returns:** None	

delete Core Group	Core Group Management group	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.	None	• Parameters: **- coreGroupName** The name of the existing core group that will be deleted. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCore Group {-coreGroupName MyCoreGroup}` • Using Jython: `AdminTask.deleteCore Group('[-coreGroup Name MyCoreGroup ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCore Group {-interactive}` • Using Jython: `AdminTask.deleteCore Group ('[-interactive ]')`
delete Core Group Access Points	Core Group Bridge Management group	The **delete Core Group Access Points** command deletes all the core group access points associated with a group that you specify.	Core group bridge settings object for the cell. (ObjectName, required)	• Parameters: **- coreGroupName** The name of the core group whose core group access points will be deleted. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCore GroupAccessPoints (cells/rohitbuildCell 01\|coregroupbridge. xml#CoreGroupBridge Settings_1) "-core GroupName Default CoreGroup"` • Using Jython: `AdminTask.deleteCore GroupAccessPoints(' (cells/rohitbuildCell 01\|coregroupbridge. xml#CoreGroupBridge Settings_1)', '[-core GroupName Default CoreGroup]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCore GroupAccessPoints {-interactive}` • Using Jython: `AdminTask.deleteCore GroupAccessPoints ('[-interactive]')`

delete SIB Destin ation	SIB Admin Comm ands	Use the **delete SIB Destination** command to delete a bus destination. This command deletes the named destination of the named bus and deletes all related message points.	None	• Parameters: **bus** name of the bus on which the destination to be deleted exists (String, required) **name** name of the destination to be deleted (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIB Destination {-bus` *busname* `-name` *destname*`}` • Using Jython: `AdminTask.deleteSIB Destination('[-bus` *busname* `-name` *destname*`]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIB Destination {-interactive}` • Using Jython: `AdminTask.deleteSIB Destination ('[-int eractive]')`

delete SIB Engine	SIB Admin Commands	Use the **delete SIB Engine** command to delete the default or named bus messaging engine from the named SIB bus. A server can only have one messaging engine, so when using this command to delete a messaging engine from a server there is no need to supply the engine name. A cluster can have more than one messaging engine so the name of the engine must be supplied.	None	• Parameters: ***bus** name of the bus to which the messaging engine to be deleted belongs (String, required) **node** to delete a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **server** to delete a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **cluster** to delete a messaging engine on a cluster, supply cluster name, but not node and server name (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIB Engine {-bus bus name -node node Name -server severname}` • Using Jython: `AdminTask.deleteSIB Engine('[-bus bus name -node node Name -server severname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIB Engine {-interactive}` • Using Jython: `AdminTask.deleteSIB Engine ('[-interac tive]')`
				engine name of the messaging engine to delete. This is optional, and is only required when deleting a messaging engine from a cluster (String, optional) • Returns: None	

delete SIB JMS Activation Spec	SIB JMS Admin Comm ands	Use the **delete SIB JMS Activation Spec** command to delete an activation specifica tion.	None	• Parameters: **name** The name of the activation specification that you want to delete. (String, (required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBJ MSActivationSpec {-name specname}` • Using Jython: `AdminTask.deleteSIBJ MSActivationSpec('[ -name specname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBJ MSActivationSpec {-interactive}` • Using Jython: `AdminTask.deleteSIBJ MSActivationSpec ('[-interactive]')`
delete SIB JMS Connec tion Factory	SIB JMS Admin Comm ands	Use the **delete SIB JMS Connection Factory** command to	None	• Parameters: **name** The name of the SIB JMS connection factory (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBJ MSConnectionFactory {-name factory_ name}` • Using Jython: `AdminTask.deleteSIBJM SConnectionFactory('[ -name factory_ name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBJ MSConnectionFactory {-interactive}` • Using Jython: `AdminTask.deleteSIBJ MSConnectionFactory ('[-interactive]')`

delete SIB JMS Queue	SIB JMS Admin Commands	Use the **delete SIB JMS Queue** command to delete a JMS queue.	None	• Parameters: **name** The name of the SIB JMS queue. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBJMSQueue {-name queue_name}` • Using Jython: `AdminTask.deleteSIBJMSQueue('[-name queue_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBJMSQueue {-interactive}` • Using Jython: `AdminTask.deleteSIBJMSQueue ('[-interactive]')`
delete SIB JMS Topic	SIB JMS Admin Commands	Use the **delete SIB JMS Topic** command to delete a JMS topic.	None	• Parameters: **name** The name of the SIB JMS topic (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBJMSTopic {-name topic_name}` • Using Jython: `AdminTask.deleteSIBJMSTopic('[-name topic_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBJMSTopic {-interactive}` • Using Jython: `AdminTask.deleteSIBJMSTopic ('[-interactive]')`
delete SIB Mediation	SIB Admin Commands	Use this command to delete the named mediation from the named bus.	None	• Parameters: **bus** name of the bus that owns the mediation (String, required) **mediationName** name of the mediation to be deleted (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBMediation {-interactive}` • Using Jython: `AdminTask.deleteSIBMediation ('[-interactive]')`

delete SIB WS Endpoint Listener	SIB Web Services group	The **delete SIB WS Endpoint Listener** command deletes the configuration oa an end point listener. This command fails if there are inbound port objects associated with the end point listener.	Object name of the end point listener that you want to delete.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SEndpointListener $epl` • Using Jython: `AdminTask.deleteSIBWS EndpointListener(epl)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SEndpointListener {-interactive}` • Using Jython: `AdminTask.deleteSIBW SEndpointListener ('[-interactive]')`
delete SIBWS Inbound Service	SIBWeb Services group	The **delete SIBWS Inbound Service** command deletes an inbound service object and any inbound port objects that are associated.	The object name of the inbound service object that you want to delete.	• Parameters: **userId** The user ID to use to interact with UDDI registries. (optional) **password** The password to use to interact with UDDI registries. (optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SInboundService $inService` • Using Jython: `AdminTask.deleteSIBWS InboundService(in Service)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SInboundService {-interactive}` • Using Jython: `AdminTask.deleteSIBWS InboundService ('[-interactive]')`

delete SIBWS Outbound Service	SIBWeb Services group	The **delete SIBWS Outbound Service** command deletes an outbound service object and any outbound port objects that are associated. Resources that are associated with the outbound service or outbound ports, for example, WS-Security configuration, are disassociated from the outbound service and the outbound ports but are not deleted.	Object name of the outbound service object that you want to delete.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SOutboundService $outService` • Using Jython: `AdminTask.deleteSIBWS OutboundService (outService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBW SOutboundService {-interactive}` • Using Jython: `AdminTask.deleteSIBW SOutboundService ('[-interactive]')`
delete SIBus	SIB Admin Comm ands	Use this command to delete the named SIB bus. Also deletes all SIB mediations and SIB destinations owned by the bus.	None	• Parameters: **bus** name of bus to be deleted from the current cell (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSI Bus {-bus bus_ name}` • Using Jython: `AdminTask.deleteSI Bus('[-bus bus_ name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSIBus {-interactive}` • Using Jython: `AdminTask.deleteSIBus ('[-interactive]')`

delete Server	None	Use the **delete Server** command to delete the server scope configuration and the server entry that corresponds to it in the `serverindex.xml` file. You can also use this command to delete a Web server.	None	• Parameters: **-nodeName** (String, required) **-serverName** (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSer ver {-nodeName node _name -serverName server_name}` • Using Jython: `AdminTask.deleteServer ('[-nodeName node_ name -serverName server_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSer ver {-interactive}` • Using Jython: `AdminTask.deleteSer ver ('[-interactive ]')`
delete Server Template	None	Use the **delete Server Template** command to delete server templates. You cannot delete templates defined by the system. You can only delete server templates that you created. This command deletes the directory that hosts the server template.	A server template identification, `javax. management. ObjectName`. This target object is required.	• Returns: Void	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSer verTemplate {-inter active}` • Using Jython: `AdminTask.deleteServ erTemplate ('[-inte ractive]')`

delete WSGW Gateway Service	WS Gateway group	The **delete WSGW Gateway Service** command deletes a gateway service. It deletes the gateway destination the corresponding reply destination, inbound service, and inbound port enablement objects, and all of the target service objects that are associated. This command does not delete the destinations that are associated with the target services.	Object name of the gateway service object	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSG WGatewayService $gwService` • Using Jython: `AdminTask.deleteWSG WGatewayService (gwService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSGW GatewayService {-interactive}` • Using Jython: `AdminTask.deleteWSG WGatewayService ('[-interactive]')`
delete WSGW Proxy Service	WS Gateway group	The **delete WSGW Proxy Service** command deletes a proxy service including the proxy destinations, outbound service, outbound ports, inbound service, and inbound port enablement objects.	Object name of the Proxy Service object	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSGW ProxyService $proxy Service` • Using Jython: `AdminTask.deleteWSGW ProxyService(proxy Service)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSGW ProxyService {-interactive}` • Using Jython: `AdminTask.deleteWSGW ProxyService ('[-interactive]')`

disconn ect SIBWS Endpoint Listener	SIBWeb Services group	The **disconnect SIBWS Endpoint Listener** command disconne cts an end point listener from a bus.	Object name of the end point listener to be disconn ected.	• Parameters: **bus** The name of the bus from which to be disconnected. (required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask disconnect SIBWSEndpointListener $epl {-bus "MyBus"}` • Using Jython: `AdminTask.disconnectS IBWSEndpointListener (epl,'[-bus MyBus ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask disconnect SIBWSEndpointListener {-interactive}` • Using Jython: `AdminTask.disconnect SIBWSEndpointListener ('[-interactive]')`
doesCore Group Exist	Core Group Manage ment group	The **does Core Group Exist** command returns a boolean value that indicates if the core group that you specify exists.	None	• Parameters: **coreGroupName** The name of the core group. (String, required) • Returns: A boolean value.	**Batch mode example usage:** • Using Jacl: `$AdminTask doesCore GroupExist {-coreGroup Name MyCoreGroup}` • Using Jython: `AdminTask.doesCore GroupExist('[-core GroupName MyCore Group]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask doesCore GroupExist {-inter active}` • Using Jython: `AdminTask.doesCore GroupExist ('[-inter active]')`

export Server	Configuration archive Operations group	Use the **export Server** command to export the server configuration to a node defined in the configuration archive. The **export Server** 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.	None	• Parameters: **-archive** The fully qualified path of the exported configuration archive. (String, required) **-nodeName** The node name of the server. This parameter is only required when the server name is not unique across the cell. (String, optional) **-serverName** The server name. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask export Server {-archive c:\myServer.ear -nodeName node1 -serverName server1}` • Using Jython: `AdminTask.exportServer('[-archive c:\myServer.ear -nodeName node1 -serverName server1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask export Server {-interactive}` • Using Jython: `AdminTask.exportServer ('[-interactive]')`

		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 SIBus membership. The **export Server** command exports the metadata file of the node where the server resides. You can use this information later when you import the configuration archive in order to verify that the target node is compatible to the node from which you are exporting the server.			

export Was profile	configuration archive Operations group	Use the **export Was profile** command to export the entire cell configuration to a configuration archive.	None	• Parameters: **archive** The fully qualified file path of the exported configuration archive. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask exportWas profile {-archive c:\myCell.ear}` • Using Jython: `AdminTask.exportWas profile('[-archive c:\myCell.ear]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask exportWas profile {-interactive}` • Using Jython: `AdminTask.exportWas profile ('[-interactive]')`
getAll Core Group Names	Core Group Management group	The **getAll Core Group Names** command returns a string that contains the names of all of the existing core groups	None	• Parameters: None • Returns: String array (String[ ])	**Batch mode example usage:** • Using Jacl: `$AdminTask getAllCoreGroupNames` • Using Jython: `AdminTask.getAllCoreGroupNames()` **Interactive mode example usage:** • Using Jacl: `$AdminTask getAllCoreGroupNames {-interactive}` • Using Jython: `AdminTask.getAllCoreGroupNames ('[-interactive]')`

getCore Group Name For Server	Core Group Management group	The **getCore Group Name For Server** command returns the name of the core group for which the server you specify is currently a member.	None	• Parameters: **- nodeName** The name of the node that contains the server. (String, required) **- serverName** The name of the server. (String, required) • Returns: The name of the core group that currently contains the server that you specified. (String)	**Batch mode example usage:** • Using Jacl: `$AdminTask getCoreGroupNameForServer {-nodeName myNode -serverName myServer}` • Using Jython: `AdminTask.getCoreGroupNameForServer('[-nodeName myNode -serverName myServer]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getCoreGroupNameForServer {-interactive}` • Using Jython: `AdminTask.getCoreGroupNameForServer ('[-interactive]')`
get Default Core Group Name	Core Group Management group	The **get Default Core Group Name** command returns the name of the default core group.	None	• Parameters: None • Returns: String	**Batch mode example usage:** • Using Jacl: `$AdminTask getDefaultCoreGroupName` • Using Jython: `AdminTask.getDefaultCoreGroupName()` **Interactive mode example usage:** • Using Jacl: `$AdminTask getDefaultCoreGroupName {-interactive}` • Using Jython: `AdminTask.getDefaultCoreGroupName ('[-interactive]')`

get Metadata Properties	Managed Object Metadata group	The **get Metadata Properties** command obtains all metadata for the node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The list of metadata properties.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMeta dataProperties {-node Name node1}` • Using Jython: `AdminTask.getMeta dataProperties(' [-nodeName node1 ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMetadata Properties {-inter active}` • Using Jython: `AdminTask.getMetadata Properties ('[-inter active]')`
get Metadata Property	Managed Object Metadata group	The **get Metadata Property** command obtains metadata with the specified key for the node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. **- propertyName** Metadata property key. • Returns: The requested property for the node that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMetadata Property {-nodeName node1 -property Name com.ibm.webs phere.baseProduct Version}` • Using Jython: `AdminTask.getMetadata Property ('[-nodeName node1 -property Name com.ibm.web sphere.baseProduct Version]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMeta dataProperty {-inter active}` • Using Jython: `AdminTask.getMetadata Property ('[-inter active]')`

get Named TCP End Point	Core Group Bridge Management group	The **get Named TCP End Point** 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.	The bridge interface object for which the port will be listed. (ObjectName, required)	• Parameters: None • Returns: The port (named end point) object name of the TCP inbound channel instance which resides on the DCS transport channel chain of the bridge interface.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNamedTCPEndPoint (cells/rohitbuildCell01\|coregroupbridge.xml#BridgeInterface_2)` • Using Jython: `AdminTask.getNamedTCPEndPoint('(cells/rohitbuildCell01\|coregroupbridge.xml#BridgeInterface_2)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNamedTCPEndPoint {-interactive}` • Using Jython: `AdminTask.getNamedTCPEndPoint ('[-interactive]')`
get Node Base Product Version	Managed Object Metadata group	The **get Node Base Product Version** command returns the version of the WebSphere Application Server for a node that you specify. This command only returns the version for a distributed installation of the product.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server version for the node that you specify.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNodeBaseProductVersion {-nodeName node1}` • Using Jython: `AdminTask.getNodeBaseProductVersion('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNodeBaseProductVersion {-interactive}` • Using Jython: `AdminTask.getNodeBaseProductVersion ('[-interactive]')`

getNode Major Version	Managed Object Metadata group	The **get Node Major Version** command returns the major version of the WebSphere Application Server for a node that you specify. It only returns the version for a distributed installation of the product.	None	• Parameters: **nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server major version for the node that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNodeMajorVersion {-nodeName node1}` • Using Jython: `AdminTask.getNodeMajorVersion('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNodeMajorVersion {-interactive}` • Using Jython: `AdminTask.getNodeMajorVersion ('[-interactive]')`
getNode Minor Version	Managed Object Metadata group	The **getNode Minor Version**command returns the minor version of the WebSphere Application Server for a node that you specify. It only returns the version for a distributed installation of the product.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server minor version for the node that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNodeMinorVersion {-nodeName node1}` • Using Jython: `AdminTask.getNodeMinorVersion('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNodeMinorVersion {-interactive}` • Using Jython: `AdminTask.getNodeMinorVersion ('[-interactive]')`

getNode Platform OS	Managed Object Metadata group	The **getNode Platform OS** command returns the operating system name for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The operating system name of the node that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNode PlatformOS {-node Name node1}` • Using Jython: `AdminTask.getNodePlat formOS('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNode PlatformOS {-inter active}` • Using Jython: `AdminTask.getNodePlat formOS ('[-interactive ]')`
getNode Sysplex Name	Managed Object Metadata group	The **getNode Sysplex Name** command returns the sysplex name for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The sysplex name of the given node.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNodeSys plexName {-nodeName node1}` • Using Jython: `AdminTask.getNodeSys plexName('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNodeSys plexName {-interac tive}` • Using Jython: `AdminTask.getNodeSys plexName ('[-intera ctive]')`
getServer Type					**Interactive mode example usage:** • Using Jacl: `$AdminTask getServer Type {-interactive}` • Using Jython: `AdminTask.getServer Type ('[-interac tive]')`

getTCP EndPoint	None	The **getTCP EndPoint** command obtains the named end point associated with either a TCP inbound channel or a chain that contains a TCP inbound channel.	TCPInbound Channel, or containing chain, instance that is associated with a NamedEnd Point. (ObjectName, required)	• Parameters: None • Returns: Object name of an existing named end point that is associated with the TCP inbound channel instance or a channel chain.	**Batch mode example usage:** • Using Jacl: `$AdminTask getTCPEnd Point TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr\|server.xml#TCP InboundChannel_1)` `$AdminTask getTCPEnd Point DCS(cells/rohit buildCell01/nodes/ rohitbuildCellManager 01/servers/dmgr|serv er.xml#Chain_3)` • Using Jython: `AdminTask.getTCPEnd Point('TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml#TCP InboundChannel_1)')` `AdminTask.getTCPEnd Point('DCS(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml# Chain_3)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getTCPEnd Point {-interactive}` • Using Jython: `AdminTask.getTCPEnd Point ('[-inter active]')`

import Server	Configuration archive Operations group	Use the **import Server** 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.	None	• Parameters: **-archive** The fully qualified path of the configuration archive. (String, required) **-nodeInArchive** 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** 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)	**Batch mode example usage:** • Using Jacl: `$AdminTask importServer {-archive c:\myServer.ear -nodeInArchive node1 -serverInArchive server1}` • Using Jython: `AdminTask.importServer('[-archive c:\myServer.ear -nodeInArchive node1 -serverInArchive server1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask importServer {-interactive}` • Using Jython: `AdminTask.importServer ('[-interactive]')`

				-nodeName The node name where the server is imported. (String, optional if there is only one node) **-serverName** 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 thrown. (String, optional, default:serverInArchive) **-coreGroup** The core group name to which the server should belong. (String, optional) • Returns: None	
help	None	The **help** command provides a summary of the help commands and ways to invoke an administrative command.	None	• Parameters: None • Returns: A general help description	• Using Jacl: `$AdminTask help` • Using Jython: `print AdminTask.help()`
help	None	The **help** command provides a list of available administrative commands if the option string is `-commands` or administrative command groups if the option string is `-commandGroups`. Valid options include `-commands` and `-commandGroups`.	None	• Parameters: **- options** • Returns: A summary of all available administrative commands.	• Using Jacl: `$AdminTask help -commands` • Using Jython: `AdminTask.help ('-commands')`

help	None	If you provide the step name, this command provides help information for a given step of an administrative command. Otherwise, it provides help information for a given admin command or administrative command group. The stepName parameter is optional.	None	• Parameters: **- commandName** **- stepName** • Returns: A summary of the specified command group, administrative command, or step.	• Using Jacl: `$AdminTask help createJ2CConnectionFactory` • Using Jython: `AdminTask.help('createJ2CConnectionFactory')`
import Was profile	configuration archive Operations group	Use the **import Was profile** command to import a cell configuration in the configuration archive to the system. Only a base single server configuration is supported for this command.	None	• Parameters: **archive** The fully qualified file path of the configuration archive. (String, required) • Returns: Void	**Batch mode example usage:** • Using Jacl: `$AdminTask importWas profile {-archive c:\myCell.ear}` • Using Jython: `AdminTask.importWas profile('[-archive c:\myCell.ear]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask importWas profile {-interactive}` • Using Jython: `AdminTask.importWas profile ('[-interactive]')`

isNode ZOS	Manged Object Metadata group	The **isNode ZOS** command tests if a node that you specify is running on the z/OS platform. This command does not apply to distributed platforms or WebSphere Application Server-Express.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: A `true` value if the node operating system is z/OS. A `false` value if the node operating system is not z/OS.	**Batch mode example usage:** • Using Jacl: `$AdminTask isNodeZOS {-nodeName node1}` • Using Jython: `AdminTask.isNodeZOS ('[-nodeName node1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask isNodeZOS {-interactive}` • Using Jython: `AdminTask.isNodeZOS ('[-interactive]')`
list Admin Object Interfaces	JCA manage ment group	Use the **list Admin Object Interfaces** command to list the administrative object interfaces defined under the resource adapter that you specify.	J2CResouce adapter object ID	• Parameters: None • Returns: A list of administrative object interfaces.	**Batch mode example usage:** • Using Jacl: `$AdminTask listAdmin ObjectInterfaces $ra` • Using Jython: `AdminTask.listAdmin ObjectInterfaces(ra)`

listChain Templates	Channel Framework Management group	The **list Chain 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.	None	• Parameters: **- 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) • Returns: A list of all the chain template object names. If you specify the acceptorFilter parameter, the list that returns is filtered to match the filter that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listChain Templates {}` `$AdminTask listChain Templates "-acceptor Filter WebContainer InboundChannel"` • Using Jython: `AdminTask.listChain Templates()` `AdminTask.listChain Templates('[-acceptor Filter WebContainer InboundChannel]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listChain Templates {-interactive}` • Using Jython: `AdminTask.listChain Templates ('[-interactive]')`

list Chains	Channel Framework Management group	The **list Chains** command lists all the chains configured under a particular instance of the transport channel service.	The instance of the transport channel service under which the the chains are configured. (ObjectName, required)	• Parameters: **- 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) • Returns: A list of all the channel chain object names that match the specified filters. If no you do not specify any parameters, all of the channel chains that are configured under the particular instance of transport channel service are returned.	**Batch mode example usage:** • Using Jacl: `$AdminTask listChains (cells/rohitbuildCell 01/nodes/rohitbuild Node01/servers/server 2\|server.xml#Transport ChannelService_1093 445762328)` `$AdminTask listChains (cells/rohitbuildCell 01/nodes/rohitbuild Node01/servers/server2 |server.xml#Transport ChannelService_10934 45762328) {-acceptor Filter *WebContain erInboundChannel*}` `$AdminTask listChains (cells/rohitbuildCell 01/nodes/rohitbuild Node01/servers/server 2\|server.xml#Transpo rtChannelService_109 3445762328) {-end PointFilter *WC_ adminhost*}` • Using Jython: `AdminTask.listChains ('(cells/rohitbuild Cell01/nodes/rohit buildNode01/servers/ server2\|server.xml# TransportChannelSer vice_1093445762328)')` `AdminTask.listChains ('(cells/rohitbuild Cell01/nodes/rohit buildNode01/servers/ server2\|server.xml# TransportChannel Service_109344576232 8)', '[-acceptorFi lter WebContainerIn boundChannel]')` `AdminTask.listChains ('(cells/rohitbuild Cell01/nodes/rohit buildNode01/servers/ server2\|server.xml# TransportChannelSer vice_1093445762328) ', '[-endPointFilter WC_adminhost]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listChains {-interactive}` • Using Jython: `AdminTask.listChains ('[-interactive]')`

list Connection Factory Interfaces	JCA management group	Use the **list Connection Factory Interfaces** command to list all of the connection factory interfaces defined under the Java 2 resource adapter that you specify.	J2C Resource Adapter object ID	• Parameters: None • Returns: A list of connection factory interfaces.	**Batch mode example usage:** • Using Jacl: `$AdminTask listConnectionFactoryInterfaces $ra` • Using Jython: `AdminTask.listConnectionFactoryInterfaces (ra)`
listCore Groups	Core Group Bridge Management group	The **listCore Groups** command returns a collection of core groups that are related to the core group that you specify.	The name of the core group for which the related core groups will be listed. (String, required)	• Parameters: **- cgBridgeSettings** The group bridge settings object for the cell. (ObjectName, required) • Returns: A set of core group names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listCoreGroups DefaultCoreGroup "-cgBridgeSettings (cells/rohitbuildCell01\|coregroupbridge.xml#CoreGroupBridgeSettings_1)"` • Using Jython: `AdminTask.listCoreGroups('DefaultCoreGroup', '[-cgBridgeSetting (cells/rohitbuildCell01\|coregroupbridge.xml#CoreGroupBridgeSettings_1)]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listCoreGroups {-interactive}` • Using Jython: `AdminTask.listCoreGroups ('[-interactive]')`

list Eligible Bridge Interfaces	Core Group Bridge Management group	The **list Eligible Bridge Interfaces** 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.	The core group access point object for which bridge interfaces will be listed. (ObjectName, required)	• Parameters: None • Returns: 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)`	**Batch mode example usage:** • Using Jacl: `$AdminTask listEligib leBridgeInterfaces CGAP_DCG_2(cells/ rohitbuildCell01\| coregroupbridge.xml# CoreGroupAccessPoint _1089636614062)` • Using Jython: `AdminTask.listEligib leBridgeInterfaces ('CGAP_DCG_2(cells/ rohitbuildCell01\| coregroupbridge.xml# CoreGroupAccessPoint_ 1089636614062)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listEligib leBridgeInterfaces {-interactive}` • Using Jython: `AdminTask.listEligib leBridgeInterfaces ('[-interactive]')`
list J2C Activation Specs	JCA management group	Use the **list J2c Activation Specs** command to list the activation specs contained under the resource adapter and message listener type that you specify.	J2C Resource Adapter object ID	• Parameters: **-messageListener Type** Specifies the message listener type for the resource adapter for which you are making a list. This parameter is required. • Returns: A list of activation specs that has specified messageListener type.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2CAct ivationSpecs $ra {-messageListenerType` *`javax.jms.Message Listener`*`}` • Using Jython: `AdminTask.listJ2CAct ivationSpecs(ra, '[-messageListener Type` *`javax.jms. MessageListener`*`]')`

list J2C Admin Objects	JCA Management group	Use the **list J2C Admin Objects** command to list administrative objects that contains the administrative object interface that you specify.	J2C Resource Adapter object ID	• Parameters: **-adminObject Interface** Specifies the administrative object interface for which you want to list. This parameter is required. • Returns: A list of administrative objects that has specified adminObjectInterface.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2C AdminObjects $ra {-adminObjectInterface fvt.adaptor.message.FVTMessageProvider}` • Using Jython: `AdminTask.listJ2C AdminObjects(ra, '[-adminObjectInterface fvt.adaptor.message.FVTMessageProvider]')`
list J2C Conne ction Factories	JCA management group	Use the **list J2C Connection Factories** command to list the Java 2 connection factories under the resource adapter and connection factory interface that you specify	J2C Resource Adapter object ID	• Parameters: **-connectionFactory Interface** Indicates the name of the connection factory that you want to list. This parameter is required. • Returns: A list of J2C connectionFactory that has the specified connectionFactoryInterface.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2C ConnectionFactories $ra {-connectionFactoryInterface javax.sql.DataSource}` • Using Jython: `AdminTask.listJ2CConnectionFactories(ra, '[-connectionFactoryInterface javax.sql.DataSource]')`
list Managed Nodes	Unmanaged Node Commands group	Use the **list Managed Nodes** command to list the managed nodes (nodes that have a node agent defined) in a configuration.	None	• Parameters: None • Returns: List	**Batch mode example usage:** • Using Jacl: `$AdminTask listManagedNodes` • Using Jython: `AdminTask.listManagedNodes()`
list Message Listener Types	JCA Management group	Use the **list Message Listener Types** command to list the message listener types defined under the resource adapter that you specify.	J2C Resource Adapter object ID	• Parameters: None • Returns: A list of message listener types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listMessageListenerTypes $ra` • Using Jython: `AdminTask.listMessageListenerTypes(ra)`

listNode Group Properties	Node Group Commands group	The **listNode Group Properties** command displays all of the custom properties of a node group.	The target object is name of the node group. This target object is required.	• Parameters: None • Returns: A list of all of the custom properties of a node group.	**Batch mode example usage:** • Using Jacl: `$AdminTask listNode GroupProperties WBINodeGroup` • Using Jython: `AdminTask.listNode GroupProperties ('WBINodeGroup')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listNode GroupProperties {-interactive}` • Using Jython: `AdminTask.listNode GroupProperties ('[-interactive]')`
listNode Groups	Node Group Commands group	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.	The target object is name of the node. This target object is optional.	• Parameters: None • Returns: A list of the node groups in the cell.	**Batch mode example usage:** • Using Jacl: `$AdminTask listNode Groups` `$AdminTask listNode Groups` *`nodeName`* • Using Jython: `AdminTask.listNode Groups` `AdminTask.listNodeGroups('`*`nodeName`*`')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listNode Groups {-interactive}` • Using Jython: `AdminTask.listNode Groups ('[-interactive]')`

list Nodes	Node Group Commands group	The **list Nodes** command displays all of the nodes in the cell.	The target object is name of the node group. This target object is optional.	• Parameters: None • Returns: A list of all the nodes in the cell	**Batch mode example usage:** • Using Jacl: `$AdminTask listNodes` • Using Jython: `AdminTask.listNodes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listNodes {-interactive}` • Using Jython: `AdminTask.listNodes ('[-interactive]')`
list SIB Destinations	SIBAdmin Commands	Use this command to get a list of SIB destinations of the named type owned by a named SIB bus. If no type is named, all destinations owned by the named bus are listed.	None	• Parameters: **bus** Bus name (String, required) **name** Destination name (String, required) **type** type of destination to list - Queue, TopicSpace, WebService or Port (String, optional) • Returns: List of SIB destinations.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIB Destinations {-bus` *`busname`* `-name` *`destname`* `-type Queue}` • Using Jython: `AdminTask.listSIB Destinations('[-bus` *`busname`* `-name` *`destname`* `-type Queue]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSIB Destinations {-interactive}` • Using Jython: `AdminTask.listSIB Destinations ('[ -interactive]')`

listSIB Engines	SIB Admin Commands	Use the **listSIB Engines** command to get a list of bus messaging engines. Supplying only the bus parameter will result in a list of all engines associated with the named bus. Supplying only the node and server parameters will result in a list of all engines owned by the named node/server. Supplying only the cluster parameter will result in a list of all engines owned by the named cluster. All other parameter combinations are illegal.	None	• Parameters: **bus** name of the bus whose engines are to be listed (String, optional) **node** node name. To list messaging engines on a server, supply node and server name, but not cluster name (String, optional) **server** server name. To list messaging engines on a server, supply node and server name, but not cluster name (String, optional) **cluster** cluster name. To list messaging engines on a cluster, supply cluster name, but not node and server name (String, optional) • Returns: A list of SIB messaging engines.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIB Engines {-bus busname -node nodeName -server severname}` • Using Jython: `AdminTask.listSIB Engines('[-bus bus name -node node Name -server severname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSIB Engines {-interactive}` • Using Jython: `AdminTask.listSIBEngi nes ('[-interactive]')`
list SIB JMS Activation Specs	SIB JMS Admin Commands				**Interactive mode example usage:** • Using Jacl: `$AdminTask listSIBJMS ActivationSpecs {-interactive}` • Using Jython: `AdminTask.listSIBJMS ActivationSpecs ('[-interactive]')`

list SIB JMS Connec tion Factories	SIB JMS Admin Comm ands	Use the **list SIB JMS Connection Factories** command to list all of the JMS connection factories for the default messaging provider at the scope that you specify.	None	• Parameters: **type** Filters the list of connection factories. Valid values include: – all - Lists all the JMS connection factories (unified, queue, and topic) at the scope that you specify. – queue - Lists all of the JMS queue connection factories at the scope that you specify. – topic - Lists all of the JMS topic connection factories at the scope that you specify. If you do no specify the type option, this command will return only the unified JMS connection factories at the scope that you specified. • Returns: A list of connection factories at the scope that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBJMS ConnectionFactories {-type queue}` • Using Jython: `AdminTask.listSIBJMS ConnectionFactories ('[-type queue]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSIBJMS ConnectionFactories {-interactive}` • Using Jython: `AdminTask.listSIBJMS ConnectionFactories ('[-interactive]')`
list SIB JMS Queues	SIB JMS Admin Comm ands	Use the **list SIB JMS Queues** command to list all the JMS queues for the default messaging provider at the specified scope.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBJM SQueues` • Using Jython: `AdminTask.listSIBJM SQueues()`

list SIB JMS Topics	SIB JMS Admin Comm ands	Lists all JMS topics for the default messaging provider at the specified scope.	None	• Parameters: None • Returns: A list of JMS topics.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBJM STopics` • Using Jython: `AdminTask.listSIBJM STopics()`
list SIB Media tions	SIB Admin Comm ands	Use this command to list the mediations on a named SIB bus.	None	• Parameters: **bus** name of the SIB bus where the mediations to be listed are to be found (String, required) • Returns: A list of SIB mediations.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBMe diations {-bus bus_name}` • Using Jython: `AdminTask.listSIBMed iations('[-bus bus_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSIBMedi ations {-interactive}` • Using Jython: `AdminTask.listSIBMedi ations ('[-interac tive]')`
list SIBus Members	SIB Admin Commands	Use this command to list all servers and clusters which are members of the named SIB bus.	None	• Parameters: **bus** name of the SIB bus whose members are to be listed (String, required) • Returns: List containing the IDs of bus members – servers and clusters.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBus Members {-bus bus_name}` • Using Jython: `AdminTask.listSIBus Members('[-bus bus_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSIBus Members {-interactive}` • Using Jython: `AdminTask.listSIBus Members ('[-inter active]')`

listSI Buses	SIB Admin Comm ands	Use this command to list all SIB buses in the cell.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask listSIBuses` • Using Jython: `AdminTask.` `listSIBuses()`
listSSL Repert oires	None	The **listSSL Repert oires** command lists all of the Secure Sockets Layer (SSL) configuration instances you can associate with an SSL inbound channel.	SSL Inbound Channel instance for which the SSLConfig candidates are listed.	• Parameters: None • Returns: A list of eligible SSL configuration object names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSSL` `Repertoires SSL_3` `(`*`cells/rohitbuild`* *`Cell01/nodes/rohit`* *`buildNode01/servers/`* *`server2\|server.xml#`* *`SSLInboundChannel_`* *`1093445762330`*`)` • Using Jython: `AdminTask.listSSLRep` `ertoires('SSL_3(` *`cells/rohitbuild`* *`Cell01/nodes/rohit`* *`buildNode01/servers`* *`/server2\|server.xml`* *`#SSLInboundChannel_`* *`1093445762330`*`)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSL` `Repertoires` `{-interactive}` • Using Jython: `AdminTask.listSSL` `Repertoires ('` `[-interactive]')`

listServer Templates	None	Use the **listServer Templates** command to query available server templates based on server type, platform, or release level.	None	• Parameters: **-serverType** (String, optional) **-platform** (String, optional) **-releaseVersion** (String, optional) • Returns: A list of server template identifications that match with the criteria that you specify with the command parameters. If you do no specify any parameters, all server templates are returned.	**Batch mode example usage:** • Using Jacl: `$AdminTask listServer Templates {-serverType server_Type}` • Using Jython: `AdminTask.listServer Templates('[-server Type server_Type]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listServer Templates {-inter active}` • Using Jython: `AdminTask.listServer Templates ('[-inter active]')`
listServer Types	None	Use the **listServer Types** command to query defined server types on a node.	The identification of a node in the cell, `javax.management.ObjectName`. This target object is optional.	• 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.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listServ erTypes {-intera ctive}` • Using Jython: `AdminTask.listServ erTypes ('[-inter active]')`
list Servers					**Interactive mode example usage:** • Using Jacl: `$AdminTask listServers {-interactive}` • Using Jython: `AdminTask.listServers ('[-interactive]')`
listTAM Settings					**Interactive mode example usage:** • Using Jacl: `$AdminTask listTAMS ettings {-interactive}` • Using Jython: `AdminTask.listTAMSe ttings ('[-inter active]')`

listTCP EndPoints	None	The **listTCP EndPoints** command lists all named end points that can be associated with a TCP inbound channel.	TCPInbound Channel instance for which named end points candidates are listed. (ObjectName, required)	• Parameters: **- excludeDistinguished** Shows only non-distinguished named end points. This parameter does not require a value. (Boolean, optional) **- unusedOnly** Shows the named end points not in use by other TCP inbound channel instances. This parameter does not require a value. (Boolean, optional) • Returns: A list of object names for the eligible named end points.	**Batch mode example usage:** • Using Jacl: `$AdminTask listTCPEnd Points TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCel lManager01/servers/ dmgr\|server.xml# TCPInboundChannel_1)` `$AdminTask listTCPEnd Points TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml# TCPInboundChannel _1) {-excludeDis tinguished}` `$AdminTask listTCPEnd Points TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml#TCP InboundChannel_1) {-excludeDistingui shed -unusedOnly}` • Using Jython: `AdminTask.listTCPEnd Points('TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml#TCP InboundChannel_1)', '[-excludeDistin guished]')` `AdminTask.listTCPEnd Points('TCP_1(cells/ rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr|server.xml# TCPInboundChannel_1) ', '[-excludeDisti nguished]')` `AdminTask.listTCPEnd Points('TCP_1(cells/ rohitbuildCell01/nod es/rohitbuildCellMan ager01/servers/dmgr| server.xml#TCPInbound Channel_1)', '[-exc ludeDistinguished -unusedOnly]')`

					Interactive mode example usage: • Using Jacl: `$AdminTask listTCPEnd Points {-interactive}` • Using Jython: `AdminTask.listTCPEnd Points ('[-inter active]')`
listTCP Thread Pools	None	The **listTCP Thread Pools** command lists all of the thread pools that can be associated with a TCP inbound channel or TCP outbound channel.	TCP Inbound Channel or TCP Outbound Channel instance for which Thread Pool candidates are listed. (Object Name, required)	• Parameters: None • Returns: A list of eligible thread pool object names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listTCPThr eadPools TCP_1(cells /rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr\|server.xml# TCPInboundChannel_1)` • Using Jython: `AdminTask.listTCPThr eadPools('TCP_1(cells /rohitbuildCell01/ nodes/rohitbuildCell Manager01/servers/ dmgr\|server.xml# TCPInboundChannel_1)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listTCPThre adPools {-interactive}` • Using Jython: `AdminTask.listTCPThre adPools ('[-inter active]')`

list Unmanaged Nodes	Unmanaged Node Commands group	Use the **list Unmanaged Nodes** command to list the unmanaged nodes in a configuration.	None	• Parameters: None • Returns: List	**Batch mode example usage:** • Using Jacl: `$AdminTask listUnmanagedNodes` • Using Jython: `AdminTask.listUnmanagedNodes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listUnmanagedNodes {-interactive}` • Using Jython: `AdminTask.listUnmanagedNodes ('[-interactive]')`
mediate SIB Destination	SIB Admin Commands	Use the **mediate SIB Destination** command to mediate a bus destination. The bus, destination, and mediation definitions must exist prior to using this command. The destination must not be mediated already.	None	• Parameters: **bus** the name of the bus where the destination is to be mediated (String, required) **destinationName** the name of the destination to be mediated (String, required) **mediationName** the name to be given to the mediation (String, required) **node** if mediating a destination to a server, specify the node and server name, but not the cluster name (String, optional) **server** if mediating a destination to a server, specify the node and server name, but not the cluster name (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask mediateSIBDestination {-bus busname -name destname -mediationName mediationName}` • Using Jython: `AdminTask.mediateSIBDestination('[-bus busname -name destname -mediationName mediationName]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask mediateSIBDestination {-interactive}` • Using Jython: `AdminTask.mediateSIBDestination ('[-interactive]')`

				cluster if mediating a destination to a cluster, specify the cluster name, but not the node or server name (String, optional) • Returns: None	
modify Node Group	Node Group Commands group	The **modify Node Group** command modifies the configuration of a node group. The node group name can not be changed. However, its short name and description are allowed. Also, its node membership can be modified.	The target object is the node group name. This target object is required.	• Parameters: **- shortName** The short name of the node group. This parameter is optional. **- description** The description of the node group. This parameter is optional. • Returns: Node group object ID.	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyNode Group WBINodeGroup {-shortName WBIGroup -description "Default node group"}` • Using Jython: `AdminTask.modifyNode Group WBINodeGroup ('[-shortName WBI Group -description "WBI node group]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyNode Group {-interactive}` • Using Jython: `AdminTask.modifyNode Group ('[-interactive]')`

modify Node Group Property	Node Group Commands group	The **modify Node Group Property** command modifies custom properties for a node group	The name of the node group. This target object is required.	• Parameters: **- 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. • Returns: Properties object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyNode GroupProperty WBINode Group {-name Channel -value "channel1"}` • Using Jython: `AdminTask.modifyNode GroupProperty('WBINode Group', '[-name Channel -value channel1]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyNode GroupProperty {-interactive}` • Using Jython: `AdminTask.modifyNode GroupProperty ('[-interactive]')`

modify SIB Destination	SIB Admin Commands	Use the **modify SIB Destination** command to modify the attributes of a SIB destination. The bus and name parameters are used to identify the SIB destination and cannot be modified.	None	• Parameters: **bus** bus name (String, required) **name** destination name (String, required) **description** description (String, optional) **reliability** the reliability quality of service for message flows through this destination, from BEST_EFFORT_NON-PERSISTENT to ASSURED_PERSISTENT, in order of increasing reliability. Higher levels of reliability have higher impacts on the performance (String, optional) **maxReliability** the maximum reliability quality of service that is accepted for values specified by producers (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB Destination {-bus busname -name destname}` • Using Jython: `AdminTask.modifySIB Destination('[-bus busname -name destname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIB Destination {-interactive}` • Using Jython: `AdminTask.modifySIB Destination ( '[-interactive]')`

				overrideOfQOS ByProducerAllowed controls the quality of service for message flows between producers and the destination. Select this option to use the quality of service specified by producers instead of the quality defined for the destination (String, optional) **defaultPriority** the default priority for message flows through this destination, in the range 0 (lowest) through 9 (highest). This default priority is used for messages that do not contain a priority value (Integer, optional) **maxFailedDeliveries** the maximum number of times that service tries to deliver a message to the destination before forwarding it to the exception destination (Integer, optional)	

				exceptionDestination the name of another destination to which the system sends a message that cannot be delivered to this destination within the specified maximum number of failed deliveries (String, optional) **sendAllowed** clear this option (setting it to false) to stop producers from being able to send messages to this destination (String, optional) **receiveAllowed** clear this option (setting it to false) to prevent consumers from being able to receive messages from this destination (String, optional) **quiesceMode** select this option (setting it to true) to indicate that the destination is quiescing. In quiesce mode, new messages for the destination cannot be added to the bus, but any messages already in the bus can still be sent to, and processed by, the destination (Boolean, optional)	

				receiveExclusive select this option (setting it to true) to allow only one consumer to attach to a destination (Boolean, optional) **topicAccessCheckRequired** topic access check required (Boolean, optional) **replyDestination** clear this option (setting it to false) to stop producers from being able to send messages to this destination (String, optional) **replyDestinationBus** clear this option (setting it to false) to prevent consumers from being able to receive messages from this destination (String, optional) **delegateAuthorization CheckToTarget** indicates whether the authorization check should be delegated to the alias or target destination (Boolean, optional)	
				• Parameters for step one: **defaultForwardRoutingPath** **bus** bus name (String, optional) **destination** destination name (String, required) • Returns: None	

modify SIB Engine	SIB Admin Commands	Use the **modify SIB Engine** command to modify the attributes of a bus messaging engine. The bus, node, server, cluster and engine parameters are used to identify the engine and cannot be modified. A server can only have one messaging engine, so when using this command to modify a messaging engine from a server there's no need to supply the engine name. However, since a cluster can have more than one messaging engine, the engine's name must be supplied.	None	• Parameters: **bus** the name of the bus to which the messaging engine is to belong (String, required) **node** to modify a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **server** to modify a messaging engine on a server, supply node and server name, but not cluster name (String, optional) **cluster** to modify a messaging engine on a cluster, supply cluster name, but not node and server name (String, optional) **engine** the name of the engine to be modified. This is only required if the engine belongs to a cluster (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB Engine {-bus bus name -node node Name -server severname}` • Using Jython: `AdminTask.modifySIB Engine('[-bus busname -node nodeName -ser ver severname]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIB Engine {-interactive}` • Using Jython: `AdminTask.modifySIB Engine ('[-inter active]')`

				description description of the messaging engine (String, optional) **initialState** whether the messaging engine is started or stopped when the associated application server is first started. Until started, the messaging engine is unavailable. (Stopped | Started) (String, optional) **destinationHighMsgs** the maximum total number of messages that the messaging engine can place on its message points (Long, optional) • Returns: None	
modify SIB JMS Activation Spec	SIB JMS Admin Commands	Use the **modify SIB JMS Activation Spec** command to modify the properties of an activation specification.	None	• Parameters: **name** The name of the activation specification that you want to modify. (String, (required) **propertyList** A list of name-value pairs. (required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSActivationSpec {-name specname -propertyList propertyList}` • Using Jython: `AdminTask.modifySIBJ MSActivationSpec(' [-name specname -propertyList propertyList]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSActivationSpec {-interactive}` • Using Jython: `AdminTask.modifySIB JMSActivationSpec ('[-interactive]')`

modify SIB JMS Connection Factory	SIB JMS Admin Commands	Use the **modify SIB JMS Connection Factory** command to modify a unified JMS connection factory at the current scope.	None	• Parameters: **name** The name of the SIB JMS connection factory. (String, required) **jndiName** The JNDI name of the SIB JMS connection factory. (String, required) **type** The type of connection factory to modify. To modify a queue connection factory, set the value to Queue. To modify a topic connection factory, set the value to Topic. If you want to create a generic connection factory, do not specify this option. (String, optional) **busName** the SIB bus name (String, optional) **category** Classifies or groups the connection factory. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSConnectionFactory {-name factory_name -jndiName jndi_name}` • Using Jython: `AdminTask.modifySIB JMSConnectionFactory ('[-name factory_name -jndiName jndi_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSConnectionFactory {-interactive}` • Using Jython: `AdminTask.modifySIB JMSConnectionFactory ('[-interactive]')`

				clientID A user-defined string. Only required for durable subscriptions. (String, optional) **connectionProximity** The proximity of acceptable messaging engines. Valid values include: Bus, Host, Cluster and Server. (String, optional) **description** The description of the connection factory. (String, optional) **durableSubscriptionHome** The durable subscription home value. (String, optional)	

				nonPersistentMapping The non-persistent mapping value. Valid values are BestEffortNonPersistent, ExpressNonPersistent, ReliableNonPersistent, ReliablePersistent, AssuredPersistent, AsSIBDestination and None. (String, optional) **password** The password that is used to modify connections from the connection factory. (String, optional) **providerEndPoints** A list of endpoint triplets separated by commas. For example: `host:port:chain` (String, optional) **readAhead** The read ahead value. Valid values include: Default, AlwaysOn, and AlwaysOff. (String, optional)	

				remoteProtocol The name of the protocol used to connect to a remote messaging engine. (String, optional) **remoteTargetGroup** (String, optional) **remoteTargetType** (String, optional) **tempQueueModelName** Temporary queue model name. (String, optional) **tempTopicModelName** Temporary topic model name. (String, optional) **userName** The user name that is used to modify connections from the connection factory. (String, optional) • Returns: None	

modify SIB JMS Queue	SIB JMS Admin Commands	Use the **modify SIB JMS Queue** command to modify a unified JMS queue at the current scope.	None	• Parameters: **name** The name of the SIB JMS queue. (String, required) **jndiName** The JNDI name of the SIB JMS queue. (String, required) **description** A description of the SIB JMS queue (String, optional) **queueName** The name of the underlying SIB queue to which the queue points (String, required) **deliveryMode** The delivery mode for messages. Legal values are ″Application″, ″NonPersistent″ and ″Persistent″ (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSQueue {-name queue_name -jndiName jndi_name -queueName queue_name}` • Using Jython: `AdminTask.modifySIB JMSQueue('[-name queue_name -jndi Name jndi_name -queueName queue _name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIBJ MSQueue {-interactive}` • Using Jython: `AdminTask.modifySIBJM SQueue ('[-inter active]')`
				timeToLive the time in milliseconds to be used for message expiration (Long, optional) **priority** the priority for messages. Whole number in the range 0 to 9 (Integer, optional) **readAhead** read-ahead value. Legal values are ″AsConnection″, ″AlwaysOn″ and ″AlwaysOff″ (String, optional) **timeToLive** (optional) • Returns: None	

modify SIB JMS Topic	SIB JMS Admin Commands	Use the **modify SIB JMS Topic** command to modify the JMS topic at the current scope.	None	• Parameters: **name** The name of the SIB JMS topic (String, required) **jndiName** the SIB JMS topic's JNDI name (String, required) **description** a description of the SIB JMS queue (String, optional) **topicSpace** the name of the underlying SIB topic space to which the topic points (String, required) ***topicName** the topic to be used inside the topic space (for example, stock/IBM) (String, required)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSTopic {-name topic_name -jndi Name jndi_name -topicName topic_ name -topicSpace topicspace_name}` • Using Jython: `AdminTask.modifySIBJ MSTopic('[-name topic_name -jndi Name jndi_name -topicName topic_ name -topicSpace topicspace_ name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIB JMSTopic {-inter active}` • Using Jython: `AdminTask.modifySIB JMSTopic ('[-inter active]')`

				deliveryMode the delivery mode for messages. Legal values are ″Application″, ″NonPersistent″ and ″Persistent″ (String, optional) **timeToLive** the time in milliseconds to be used for message expiration (Long, optional) **priority** the priority for messages. Whole number in the range 0 to 9 (Integer, optional) **readAhead** read-ahead value. Legal values are ″AsConnection″, ″AlwaysOn″ and ″AlwaysOff″ (String, optional) **busName** the name of the bus on which the topic resides (String, optional) • Returns: None	

modify SIB Mediation	SIB Admin Commands	Use this command to modify the attributes of a SIB mediation. The bus and mediation Name parameters identify the mediation and cannot be modified.	None	• Parameters: **bus** the name of the bus that owns the mediation (String, required) **mediationName** name of the mediation to be modified (String, required) **description** description of the mediation (String, optional) **handlerListName** the name of the handler list that was defined when the mediation was deployed (String, optional) **globalTransaction** whether or not a global transaction is started for each message processed (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB Mediation {-bus bus_name -jndi Name jndi_name}` • Using Jython: `AdminTask.modifySIB Mediation('[-bus bus_name -media tionName media tion_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySIBM ediation {-inter active}` • Using Jython: `AdminTask.modifySIB Mediation ('[-inter active]')`

				allowConcurrentMediation whether or not to apply the mediation to multiple messages concurrently, and preserve message ordering (Boolean, optional) **selector** the text string that must be present in a message for the mediation to process the message (String, optional) **discriminator** the text string that must not be present in a message for the mediation to process the message (String, optional) • Returns: None	

modify SIBus	SIB Admin Commands	Use this command to modify the attributes of the named bus.He “bus” parameter identifies the bus to be modified, and is not used to change the name of the bus.	None	• Parameters: **bus** name of bus to modify (String, required) **description** description of bus modify (String, optional) **secure** enable or disable bus security (Boolean, optional) **interEngineAuthAlias** name of the authentication alias used to authorize communication between messaging engines on the bus. **mediationsAuthAlias** name of the authentication alias used to authorize mediations to access the bus (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySI Bus {-bus busname -description text -secure True -mediat ionsAuthAlias name -protocol protocol -disca rdOnDelete False}` • Using Jython: `AdminTask.modifySIBus ('[-busbusname -description "text" -secure True -mediationsAuth Alias name -protocol proto col -discardOn Delete False]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySI Bus {-interactive}` • Using Jython: `AdminTask.modifySI Bus ('[-inter active]')`

				protocol the protocol used to send and receive messages between messaging engines, and between API clients and messaging engines (String, optional) **discardOnDelete** indicate whether or not any messages left in a queue's data store should be discarded when the queue is deleted (Boolean, optional) **destinationHighMsgs** the maximum number of messages that any queue on the bus can hold (Long, optional) **configurationReloadEnabled** indicate whether configuration files should be dynamically reloaded for this bus (Boolean, optional) • Returns: None	

modify SIBus Member	SIB Admin Commands	Use this command to modify the attributes of the bus member identified by the bus, node, server and cluster parameters.	None	• Parameters: **bus** name of bus to which the member belongs(String, required) **node** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **server** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **cluster** to specify a cluster bus member, supply cluster name but not node and server name (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySIB usMember {-bus busname -node nodename -ser ver servername -description text}` • Using Jython: `AdminTask.modifySI BusMember('[-bus busname -node nodename -ser ver servername -description "text"]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySI BusMember {-inter active}` • Using Jython: `AdminTask.modifySIB usMember ('[-inter active]')`

move Cluster ToCore Group	Core Group Manage ment group	The **move Cluster ToCore Group** command moves all of servers in a cluster that you specify from a core group to another core group. All of the servers in cluster must be members of the same core group.	None	• Parameters: **- 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) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask moveClus terToCoreGroup {-source OldCore Group -target NewCoreGroup -clusterName ClusterOne}` • Using Jython: `AdminTask.moveClus terToCoreGroup(' [-source OldCore Group -target NewCoreGroup -clusterName ClusterOne]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask moveClu sterToCoreGroup {-interactive}` • Using Jython: `AdminTask.moveClus terToCoreGroup ('[-interactive]')`

moveServerToCoreGroup	Core Group Management group	The **moveServerToCoreGroup** command moves a server to a core group that you specify. When the server is added to the core group that you specify, it will be removed from the core group where it originally resided.	None	• Parameters: **- 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) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask moveServerToCoreGroup {-source OldCoreGroup -target NewCoreGroup -nodeName myNode -serverName myServer}` • Using Jython: `AdminTask.moveServerToCoreGroup('[-source OldCoreGroup -target NewCoreGroup -nodeName myNode -serverName myServer]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask moveServerToCoreGroup {-interactive}` • Using Jython: `AdminTask.moveServerToCoreGroup ('[-interactive]')`

publish SIBWS Inbound Service	SIB Web Services group	The publish SIBWS Inbound Service command publishes the WSDL document for the inbound service and the associated ports to the registry and business defined by the UDDIPublication object.	The object name of the inbound service object.	• Parameters: **uddiPublication** The name of the UDDI publication for the service. (required) **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask publishSIB WSInboundService $inService {-uddi Publication "MyUddi"}` • Using Jython: `AdminTask.publishSIB WSInboundService(in Service, '[-uddiPub lication MyUddi ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask publishSI BWSInboundService {-interactive}` • Using Jython: `AdminTask.publishSIB WSInboundService ('[-interactive]')`
reconfigure TAM					**Interactive mode example usage:** • Using Jacl: `$AdminTask reconfigur eTAM {-interactive}` • Using Jython: `AdminTask.reconfigur eTAM ('[-intera ctive]')`

refresh SIBWS Inbound Service WSDL	SIB Web Services group	The **refresh SIBWS InboundService WSDL** command loads the WSDL document from the WSDL Location parameters of the inbound service and locates the WSDL Location-specified service element. If the service element is not present, this command fails. If the outbound ports are not a subset of the ports in the loaded WSDL document, this command fails. If the WSDL will be retrieved through a proxy, the server on which the command is running must have the system properties that identify the proxy server set correctly.	The object name of the inbound service object.	• Parameters: **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask refreshSIB WSInboundServiceWSDL $inService` • Using Jython: `AdminTask.refreshSI BWSInboundService WSDL(inService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask refreshSIB WSInboundServiceWSDL {-interactive}` • Using Jython: `AdminTask.refreshSIB WSInboundServiceWSDL ('[-interactive]')`

refresh SIBWS Outbound Service WSDL	SIB Web Services group	The **refresh SIBWS Outbound Service WSDL** command loads the WSDL document from the WSDLLocation parameters of the outbound service and locates the WSDLLocation specified service element. If the service element is not present, this command fails. If the outbound ports are not a subset of the ports in the loaded WSDL document, this command fails.	The object name of the outbound service object.	• Parameters: **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask refreshSIB WSOutboundServiceWS DL $outService` • Using Jython: `AdminTask.refreshSIB WSOutboundServiceWS DL(outService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask refreshSI BWSOutboundService WSDL {-interactive}` • Using Jython: `AdminTask.refreshSIB WSOutboundServiceWSDL ('[-interactive]')`
		If the WSDL will be retrieved through a proxy, the server on which the command is running must have the system properties that identify the proxy server set correctly.			

remove Node Group	Node Group Commands group	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 can not be removed.	The name of the node group to be removed. This target object is required.	• Parameters: None • Returns: Node group object ID.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeNodeGroup WBINodeGroup` • Using Jython: `AdminTask.removeNodeGroup('WBINodeGroup')` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeNodeGroup {-interactive}` • Using Jython: `AdminTask.removeNodeGroup ('[-interactive]')`
remove Node Group Member	Node Group Commands group	The **remove Node 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.	The target object is the node group containing the member to be removed. This target object is required.	• Parameters: **- nodeName** The name of the node to be removed from a node group. This parameter is required. • Returns: Node group member object ID.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeNodeGroupMember WBINodeGroup {-nodeName WBINode}` • Using Jython: `AdminTask.removeNodeGroupMember('WBINodeGroup', '[-nodeName WBINode]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeNodeGroupMember {-interactive}` • Using Jython: `AdminTask.removeNodeGroupMember ('[-interactive]')`

remove Node Group Property	Node Group Commands group	The **remove Node Group Property** command removes custom properties of a node group.	The name of the node group. This target object is required.	• Parameters: **- name** The name of the custom property to remove. This parameter is required. • Returns: Properties object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask removeNode GroupProperty WBINode Group {-name Channel}` • Using Jython: `AdminTask.removeNode GroupProperty('WBI NodeGroup', '[-name Channel]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeNode GroupProperty {-interactive}` • Using Jython: `AdminTask.removeNode GroupProperty ('[-interactive]')`
remove SIBWS Inbound Port	SIBWeb Services group	The **remove SIBWS Inbound Port** command removes the configuration of an inbound port.	The object name of the inbound port object that you want to remove.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeSIB WSInboundPort $inPort` • Using Jython: `AdminTask.removeSIBW SInboundPort(inPort)` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeSIBW SInboundPort {-inte ractive}` • Using Jython: `AdminTask.removeSIB WSInboundPort ('[ -interactive]')`

remove SIBWS Outbound Port	SIB Web Services group	The remove SIBWS Outbound Port command removes the configur ation of an outbound port. If the port that you delete is the default port for the outbound service, one of the remaining ports, if any, will be chosen as the new default. Resources that are associated with the outbound port, for example, WS-Security configuration, are disassociated from the outbound port but not deleted.	Object name of the outbound port object that you want to remove.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeSIB WSOutboundPort $outPort` • Using Jython: `AdminTask.removeSIB WSOutboundPort (outPort)` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeSIB WSOutboundPort {-interactive}` • Using Jython: `AdminTask.removeSIBW SOutboundPort (' [-interactive]')`

remove SIBus Member	SIB Admin Commands	Use this command to remove a server or a cluster from a SIB bus. As well as removing the server or cluster from the SIB bus, this command also deletes all SIB messaging engines associated with the bus, all queue points and publication points owned by those engines, and all queue point references and publication point references which refer to the deleted queue points and publication points.	None	• Parameters: **bus** name of SIB bus to remove member from (String, required) **node** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **server** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **cluster** to specify a cluster bus member, supply cluster name but not node and server name (String, optional) • Returns:	**Batch mode example usage:** • Using Jacl: `$AdminTask removeSIBusMember {-bus busname -node nodename -server servername}` • Using Jython: `AdminTask.removeSIBusMember('[-bus busname -node nodename -server servername ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeSIBusMember {-interactive}` • Using Jython: `AdminTask.removeSIBusMember ('[-interactive]')`

remove Unmanaged Node	Unmanaged Node Commands group	Use the **remove Unmanaged Node** command to remove an unmanaged node from the configuation.	None	• Parameters: **- nodeName** The name of the unmanaged node. (String, required) • Returns: null	**Batch mode example usage:** • Using Jacl: `$AdminTask removeUnmanagedNode {-nodeName myNode }` • Using Jython: `AdminTask.removeUnmanagedNode('[-nodeName myNode]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeUnmanagedNode {-interactive}` • Using Jython: `AdminTask.createUnmanagedNode ('[-interactive]')`
remove WSGW Target Service	WS Gateway group	The **remove WSGW Target Service** command removes a target service from the gateway service. The destinations that are associated with the target service are not deleted. If the target service that you remove is the default target service, the default is set to the first target service in the set or cleared if there are none left.	object name of the Target Service object	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeWSGWTargetService $gwTarget` • Using Jython: `AdminTask.removeWSGWTargetService (gwTarget)` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeWSGWTargetService {-interactive}` • Using Jython: `AdminTask.removeWSGWTargetService ('[-interactive]')`

set Default SIBWS Outbound Port	SIB Web Services group	The **set Default SIBWS Outbound Port** command updates the default outbound port for an outbound service.	The object name of the outbound service whose default port you want to update.	• Parameters: **name** The name of the port that you want to set as the default. (required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setDefaul tSIBWSOutboundPort $outService {-name "MyServiceSoap"}` • Using Jython: `AdminTask.setDefault SIBWSOutboundPort (outService, '[-name MyServiceSoap]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask setDefault SIBWSOutboundPort {-interactive}` • Using Jython: `AdminTask.setDefault SIBWSOutboundPort ('[-interactive]')`
show SIB Destina tion	SIB Admin Comm ands	Use the **show SIB Destin ation** command to get the attribute names/ values of a SIB destination. The bus and name parameter identify the SIB destination whose attributes are required.	None	• Parameters: **bus** bus name (String, required) **name** destination name (String, required) • Returns: The attribute names and values of the named SIB destination on the named bus.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIB Destination {-bus` *`busname`* `-name` *`destname`*`}` • Using Jython: `AdminTask.showSIBDes tination('[-bus` *`busname`* `-name` *`destname`*`]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBDes tination {-inter active}` • Using Jython: `AdminTask.showSIBDes tination ('[-inter active]')`

showSIB Engine	SIB Admin Commands	Use the **showSIB Engine** command to get the attribute names/values of a SIB messaging engine belonging to a given bus member. If the bus member is a server, only the bus, node and server parameters need be supplied. A server only has 1 engine, so the engine parameter is not necessary. If the bus member is a cluster, the bus, cluster and engine parameters must be supplied, since a cluster can have more than one engine.	None	• Parameters: **bus** the name of the bus to which the messaging engine to be shown belongs (String, required) **node** to show a messaging engine that belongs to a server, supply node and server name, but not cluster name (String, optional) **server** to show a messaging engine that belongs to a server, supply node and server name, but not cluster name (String, optional) **cluster** to show a messaging engine that belongs to a cluster, supply cluster name, but not node and server name (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBEng ine {-bus busname -node nodeName -server severname}` • Using Jython: `AdminTask.showSIBEng ine('[-bus busname -node nodeName -server severname ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBEn gine {-interactive}` • Using Jython: `AdminTask.showSIBEn gine ('[-interac tive]')`

				engine The name of the engine to show. If the bus member has only one messaging engine, you do not need to specify the engine option. If the bus member has several messaging engines, you must specify the name of the engine for which you want to display details. (String, optional) • Returns: The attribute names and values of the identified SIB messaging engine.	
show SIB JMS Activation Spec	SIB Admin Commands	The showSIB JMS Activation Spec command shows details about a JMS activation specification.	None	• Parameters: **bus** The name of the bus that owns the mediation (String, required) **mediationName** The name of the mediation to be shown (String, required) • Returns: A list	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBJMS ActivationSpec {-bus bus_name -mediationName mediation_name}` • Using Jython: `AdminTask.showSIBJMS ActivationSpec('[-bus bus_name -mediationName mediation_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBJMS ActivationSpec {-interactive}` • Using Jython: `AdminTask.showSIBJMS ActivationSpec ('[-interactive]')`

show SIBJMS Connec tion Factory	SIBJMS Admin Comm ands	The **show SIBJMS Conne ction Factory** command shows details about a JMS connection factory.	None	• Parameters: **name** The name of the SIB JMS connection factory (String, required) • Returns: A set of property value pairs for the JMS connection factory that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBJMS ConnectionFactory {-name factory_name}` • Using Jython: `AdminTask.showSIBJMS ConnectionFactory(' [-name factory_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBJMS ConnectionFactory {-interactive}` • Using Jython: `AdminTask.showSIBJMS ConnectionFactory ('[-interactive]')`
show SIBJMS Queue	SIB JMS Admin Comm ands	Use the **show SIBJMS Queue** command to show the details about a JMS queue.	None	• Parameters: **name** The name of the SIB JMS queue. (String, required) • Returns: A set of property value pairs for the JMS queue that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBJMS Queue {-name queue_name}` • Using Jython: `AdminTask.showSIBJMS Queue('[-name queue_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBJMS Queue {-interactive}` • Using Jython: `AdminTask.showSIBJMS Queue ('[-intera ctive]')`

show SIBJMS Topic	SIBJMS Admin Commands	Use this command to show the details for a JMS topic.	None	• Parameters: **- name** The name of the SIB JMS topic (String, required) • Returns: A set of property value pairs for the JMS topic that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBJMSTopic {-name topic_name}` • Using Jython: `AdminTask.showSIBJMSTopic('[-name topic_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBJMSTopic {-interactive}` • Using Jython: `AdminTask.showSIBJMSTopic ('[-interactive]')`
show SIB Mediation	SIB Admin Commands	Use this command to get the attribute names/ values of a SIB mediation.	None	• Parameters: **bus** the name of the bus that owns the mediation (String, required) **mediationName** the name of the mediation to be shown (String, required) • Returns: The attribute names and values of the identified SIB mediation.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBMediation {-bus bus_name -mediationName mediation_name}` • Using Jython: `AdminTask.showSIBMediation('[-bus bus_name -mediationName mediation_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBMediation {-interactive}` • Using Jython: `AdminTask.showSIBMediation ('[-interactive]')`

show SIBus	SIB Admin Commands	Use this command to get the attribute names/ values of a SIB bus.	None	• Parameters: **bus** bus name (String, required) • Returns: The attribute names and values of the identified SIB bus.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBus {-bus bus_name}` • Using Jython: `AdminTask.showSIBus ('[-bus bus_name ]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBus {-interactive}` • Using Jython: `AdminTask.showSIBus ('[-interactive]')`
show SIBus Member	SIB Admin Comm ands	Use this command to get the attribute names/ values of a SIB bus member.	None	• Parameters: **bus** name of bus to show member from (String, required) **node** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **server** to specify a server bus member, supply node and server name, but not cluster name (String, optional) **cluster** to specify a cluster bus member, supply cluster name but not node and server name (String, optional) • Returns: The attribute names and values of the identified SIB bus member.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSIBus Member {-bus busname -node nodename -server servername}` • Using Jython: `AdminTask.showSIBus Member('[-bus busname -node nodename -server servername]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSIBus Member {-interactive}` • Using Jython: `AdminTask.showSIBus Member ('[-inter active]')`

show Server Info					**Interactive mode example usage:** • Using Jacl: `$AdminTask showServer Info {-interactive}` • Using Jython: `AdminTask.showServer Info ('[-intera ctive]')`
show Server Type Info					**Interactive mode example usage:** • Using Jacl: `$AdminTask showServer TypeInfo {-intera ctive}` • Using Jython: `AdminTask.showServer TypeInfo ('[-intera ctive]')`
show Template Info	None	Use the **show Template Info** command to query metadata information for a specific template. This command will only work for server templates.	The identification of a server template, `javax.management.ObjectName`. This target object is required.	• Returns: A property object that holds the metadata information regarding a specific template.	**Interactive mode example usage:** • Using Jacl: `$AdminTask showTempl ateInfo {-interactive}` • Using Jython: `AdminTask.showTempl ateInfo ('[-intera ctive]')`
unconfigure TAM					**Interactive mode example usage:** • Using Jacl: `$AdminTask unconfigur eTAM {-interactive}` • Using Jython: `AdminTask.unconfigur eTAM ('[-interact ive]')`

unmedi ate SIB Destination	SIB Admin Commands	Use this command to unmediated the named destination on the named bus. Unmediating a destination simply removes the association between a SIB destination and a SIB mediation.	None	• Parameters: **bus** the name of the bus where the destination is currently mediated (String, required) **destinationName** the name of the destination to be unmediated (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask unmediate SIBDestination {-bus bus_name -destinationName destination_name}` • Using Jython: `AdminTask.unmediateSIBDestination('[-bus bus_name -destinationName destination_name]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask unmediate SIBDestination {-interactive}` • Using Jython: `AdminTask.unmediateSIBDestination ('[-interactive]')`
unpublish SIBWS Inbound Service	SIBWeb Services group	The **unpublish SIBWS Inbound Service** command removes the WSDL document for the inbound service, including the ports, from the registry and business defined by the UDDI publication object.	The object name of the inbound service object.	• Parameters: **uddiPublication** The name of the UDDI publication for the service. (required) **userId** The user ID to use to retrieve the WSDL. (optional) **password** The password to use to retrieve the WSDL. (optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask unpublish SIBWSInboundService $inService {-uddiPublication "MyUddi"}` • Using Jython: `AdminTask.unpublishSIBWSInboundService(inService, '[-uddiPublication MyUddi]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask unpublish SIBWSInboundService {-interactive}` • Using Jython: `AdminTask.unpublishSIBWSInboundService ('[-interactive]')`

update App On Cluster	None	The **update AppOn Cluster** command can be used to synchronize nodes and restart cluster members for an application update deployed to a cluster. After 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 first stopping the cluster members on which the application is targetted and then performing nodesync operation and then restarting the cluster members.	None	• Parameters: **-Application Names** The names of the applications that are updated. **-timeout** The timeout value in seconds for each node synchronization. The default is 300 seconds. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateApp OnCluster {-Applica tionNames app1}` `$AdminTask updateApp OnCluster { -Applic ationNames app1 -timeout 600}` • Using Jython: `AdminTask.updateApp OnCluster('[-Applica tionNames app1]')` `AdminTask.updateApp OnCluster('[-Applic ationNames app1 -timeout 600]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateApp OnCluster -interactive` • Using Jython: `AdminTask.updateApp OnCluster ('[-inte ractive]')`

		This command may take more 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, when a SOAP connector is used, and in the sas.client.props file, when a RMI connector is used. This command is not supported in local mode.			

Administrative command invocation syntax

You can use an administrative command in batch mode or interactive mode. The following is the syntax for using an administrative command:

Using Jacl:

```
$AdminTask cmdName [targetObject] [options]
```

where `options` include:

```
{
  [-paramName paramValue] [-paramName] ...
  [-stepName {{stepParamValue ...} ...} ...]
  [-delete {-stepName {{stepKeyParamValue ...} ...} ...} ...]
  [-interactive]
}
```

Using Jython:

```
AdminTask.cmdName(['targetObject'], [options])
```

where `options` include:

```
'[
[-paramName paramValue] [-paramName ...]
[-stepName [[stepParamValue ...] ...] ...]
[-delete [-collectionStepName [[stepKeyParamValue ...] ...] ...] ...]
[-interactive]
]'
```

where:

cmdName	represents the name of an administrative command to be 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 executed command. 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.
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, then 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 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 allows the addition of new objects, the input values are added.
delete	represents the option to delete existing data from specified step that supports collection.
collectionStepName	represents the collection step name .
stepKeyParamValue ...	represents the values of key parameters to uniquely identify an object to be deleted 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.

Properties used by scripted administration

This article explains the Java properties that are used by scripted administration. Three levels of default property files load before any property file that is specified on the command line. The first level represents an installation default, located in the properties directory for each WebSphere 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. These 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.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, RMI, or NONE. The wsadmin.properties file specifies SOAP as the connector.

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.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.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.traceString
Turns on tracing for the scripting process. The default has tracing turned off.

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 logs directory of each WebSphere 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.validationOutput
Determines where the validation reports are directed. The default file is wsadmin.valout which is located in the logs directory of each WebSphere 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.

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.tempdir
Determines the directory to use for temporary files when installing applications.

The Java virtual machine 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.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.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.

Chapter 6. Using Ant to automate tasks

To support using Apache Ant with Java 2 Platform, Enterprise Edition (J2EE) applications running on the application server, the product provides a copy of the Ant tool and a set of Ant tasks that extend the capabilities of Ant to include product-specific functions. Ant has become a very popular tool among Java programmers.

Apache Ant is a Java-based build tool. In theory, it is similar to Make, but Ant is different. Instead of a model in which it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, XML-based configuration files are used. These files reference a target tree in which various tasks are run. Each task is run by an object that implements a particular Task interface.

By combining the following tasks with those provided by Ant, you can create build scripts that compile, package, install, and test your application on the application server:

- Install and uninstall applications
- Start and stop servers in a base configuration
- Run administrative scripts or commands
- Run the Enterprise JavaBeans (EJB) deployment tool
- Run the JavaServer Pages (JSP) file precompilation tool

For more detailed information about Ant, refer to the Apache organization Web site.

- To run Ant and have it automatically see the WebSphere classes, use the “ws_ant command.”
- Use “Ant tasks for deployment and server operation.”

  This topic describes where to find the API documentation for the Apache Ant tasks for deploying applications and operating application servers.
- Use “Ant tasks for building application code” on page 514.

  This topic describes where to find the API documentation for the Apache Ant tasks for building applications.

ws_ant command

This topic describes where to find information about the ws_ant command, which is provided for using with Apache Ant, a Java-based build tool that is popular among Java programmers.

In theory, Ant is similar to Make, but Ant is different. Instead of a model in which it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, XML-based configuration files are used. These files reference a target tree in which various tasks are run. Each task is run by an object that implements a particular Task interface.

For the Apache Ant tool that is provided by this product, see the following file:

install_root/bin/ws_ant.bat|sh

Ant tasks for deployment and server operation

This topic describes where to find the API documentation for the Apache Ant tasks for deploying applications and operating application servers.

The Apache Ant tasks for the product reside in the Java package: `com.ibm.websphere.ant.tasks`. The API documentation for this package contains detailed information about all of the Ant tasks that are provided and how to use them. The API documentation is available in the **Reference** section of the information center.

© Copyright IBM Corp. 2004

Ant tasks for building application code

This topic describes where to find the documentation for the Ant tasks provided for building application code using the Application Server Toolkit (which is a CD included with WebSphere Application Server as a separately installable toolkit).

Note that this toolkit includes an Automated Deployment example ″Example: Automated Deploy″ for JACL scripted deployment of multiple application updates to multiple servers and clusters in a WebSphere Network Deployment cell.

Within the Application Server Toolkit product documentation, open the section **Working with Ant**. You can locate the topic by searching for **Working with Ant**, or from the navigation view, select **Help > Help Contents > Developing Java Applications > Developing enterprise applications > J2EE applications > Working with Ant**.

Chapter 7. Using administrative programs (JMX)

This topic describes how to use Java application programming interfaces (APIs) to administer WebSphere Application Server and to manage your applications.

You can administer WebSphere Application Server and your applications through tools that come with the product or through programming with the Java APIs.

The wsadmin scripting tool, the administrative console, and the administrative command-line tools come with the product. These administrative tools provide most of the functions that you need to manage the product and the applications that run in WebSphere Application Server. You can use the command-line tools from automation scripts to control the servers. Scripts that are written for the wsadmin scripting tool offer a wide range of possible custom solutions that you can develop quickly.

Investigate these tools with the Java APIs to determine the best ways to administer WebSphere Application Server and your applications. For information on the Java APIs, view Java Management Extensions (JMX) API documentation.

WebSphere Application Server supports access to the administrative functions through a set of Java classes and methods. You can write a Java program that performs any of the administrative features of the WebSphere Application Server administrative tools. You can also extend the basic WebSphere Application Server administrative system to include your own managed resources.

You can prepare, install, uninstall, edit, and update applications through programming. Preparing an application for installation involves collecting various types of WebSphere Application Server-specific binding information to resolve references that are defined in the application deployment descriptors. This information can also be modified after installation by editing a deployed application. Updating consists of adding, removing or replacing a single file or a single module in an installed application, or supplying a partial application that manipulates an arbitrary set of files and modules in the deployed application. Updating the entire application uninstalls the old application and installs the new one. Uninstalling an application removes it entirely from the WebSphere Application Server configuration.

Perform any or all of the following tasks to manage WebSphere Application Server and your Java 2 Platform, Enterprise Edition (J2EE) applications through programming.

- Create a custom Java administrative client program using the Java administrative APIs.

  This topic describes how to develop a Java program that uses the WebSphere Application Server administrative APIs to access the administrative system of WebSphere Application Server.
- Extend the WebSphere Application Server administrative system with custom MBeans.

  This topic describes how to extend the WebSphere Application Server administration system by supplying and registering new JMX MBeans in one of the Application Server processes. In this case, you can use the administrative classes and methods to add newly managed objects to the administrative system.
- Deploy and manage a custom Java administrative client program for use with multiple Java 2 Platform, Enterprise Edition application servers.

  This topic describes how to connect to a J2EE server, and how to manage multiple vendor servers.
- Manage applications through programming

  This topic describes how, through Java MBean programming, to install, update, and delete a J2EE application on WebSphere Application Server.

Depending on which tasks you complete, you have created your own administrative program, extended the WebSphere Application Server administrative console, connected and managed vendor servers, or managed your applications through programming.

© Copyright IBM Corp. 2004

You can continue to administer WebSphere Application Server and your applications through programming or in combination with the tools that come with the WebSphere Application Server.

Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs

This section describes how to develop a Java program for accessing the WebSphere Application Server administrative system by using the WebSphere Application Server administrative application programming interfaces (APIs).

This task assumes a basic familiarity with Java Management Extensions (JMX) API programming. See the JMX API documentation for information.

When you develop and run administrative clients that use various JMX connectors and that have security enabled, use the following guidelines. When you follow these guidelines, you guarantee the behavior among different implementations of JMX connectors. Any programming model that strays from these guidelines is unsupported.

1. Create and use a single administrative client before you create and use another administrative client.
2. Create and use an administrative client on the same thread.
3. Use one of the following ways to specify a user ID and password to create a new administrative client:
   - Specify a default user ID and password in the property file.
   - Specify a user ID and password other than the default. Once you create an administrative client with a non-default user ID and password, specify the non-default user ID and password when you create subsequent administrative clients.

1. Develop an administrative client program.
2. Build and run the administrative client program.

   The steps required to build and run your program depends on the kind of application environment your code runs. Refer to ″Using application clients″ for details on how to build and run your administrative client program.

Developing an administrative client program

This page contains examples of key features of an administrative client program that utilizes WebSphere Application Server administrative APIs and Java Management Extensions (JMX). WebSphere Application Server administrative APIs provide control of the operational aspects of your distributed system as well as the ability to update your configuration. This page also demonstrates examples of operational control. For information, view the Administrative API documentation, the JMX API documentation, or the MBean API documentation.

1. Create an AdminClient instance. An administrative client program needs to invoke methods on the AdminService object that is running in the deployment manager (or the application server in the base installation). The AdminClient class provides a proxy to the remote AdminService object through one of the supported Java Management Extensions (JMX) connectors. The following example shows how to create an AdminClient instance:

```
Properties connectProps = new Properties();
connectProps.setProperty(
AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);

connectProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
connectProps.setProperty(AdminClient.CONNECTOR_PORT, "8879");
AdminClient adminClient = null;
try
{
       adminClient = AdminClientFactory.createAdminClient(connectProps);
}
```

```
catch (ConnectorException e)
{
       System.out.println("Exception creating admin client: " + e);
}
```

2. Find an MBean Once you obtain an AdminClient instance, you can use it to access managed resources in the administration servers and application servers. Each managed resource registers an MBean with the AdminService through which you can access the resource. The MBean is represented by an ObjectName instance that identifies the MBean. An ObjectName consists of a domain name followed by an unordered set of one or more key properties. For the WebSphere Application Server, the domain name is `WebSphere` and the key properties defined for administration are as follows:

type	The type of MBean. For example: Server, TraceService, Java Virtual Machine (JVM).
name	The name identifier for the individual instance of the MBean.
cell	The name of the cell that the MBean is running.
node	The name of the node that the MBean is running.
process	The name of the process that the MBean is running.

   You can locate an MBean by querying for them with ObjectNames that match desired key properties. The following example shows how to find the MBean for the NodeAgent of node MyNode:

```
String nodeName = "MyNode";
String query = "WebSphere:type=NodeAgent,node=" + nodeName + ",*";
ObjectName queryName = new ObjectName(query);
ObjectName nodeAgent = null;
Set s = adminClient.queryNames(queryName, null);
if (!s.isEmpty())
    nodeAgent = (ObjectName)s.iterator().next();
else
    System.out.println("Node agent MBean was not found");
```

3. Use the MBean. What a particular MBean allows you to do depends on that MBean's management interface. It may declare attributes that you can obtain or set. It may declare operations that you can invoke. It may declare notifications for which you can register listeners. For the MBeans provided by the WebSphere Application Server, you can find information about the interfaces they support in the MBean API documentation. The following example invokes one of the operations available on the NodeAgent MBean that we located above. The following example will start the *MyServer* application server:

```
String opName = "launchProcess";
String signature[] = { "java.lang.String" };
String params[] = { "MyServer" };
try
{
     adminClient.invoke(nodeAgent, opName, params, signature);
}
catch (Exception e)
{
     System.out.println("Exception invoking launchProcess: " + e);
}
```

4. Register for events. In addition to managing resources, the Java Management Extensions (JMX) API also supports application monitoring for specific administrative events. Certain events produce notifications, for example, when a server starts. Administrative applications can register as listeners for these notifications. The WebSphere Application Server provides a full implementation of the JMX notification model, and provides additional function so you can receive notifications in a distributed environment. For a complete list of the notifications emitted from WebSphere Application Server MBeans, refer to the `com.ibm.websphere.management.NotificationConstants` class in the API documentation. The following is an example of how an object can register itself for event notifications emitted from an MBean using the node agent ObjectName:

```
adminClient.addNotificationListener(nodeAgent, this, null, null);
```

In this example, the `null` value will result in receiving all of the node agent MBean event notifications. You can also use the `null` value with the `handback` object.

5. Handle the events. Objects receive JMX event notifications via the handleNotification method which is defined by the NotificationListener interface and which any event receiver must implement. The following example is an implementation of handleNotification that reports the notifications that it receives:

```
public void handleNotification(Notification n, Object handback)
{
     System.out.println("************************************************");
     System.out.println("* Notification received at " + new Date().toString());
     System.out.println("* type      = " + ntfyObj.getType());
     System.out.println("* message   = " + ntfyObj.getMessage());
     System.out.println("* source    = " + ntfyObj.getSource());
     System.out.println(
     "* seqNum    = " + Long.toString(ntfyObj.getSequenceNumber()));
     System.out.println("* timeStamp = " + new Date(ntfyObj.getTimeStamp()));
     System.out.println("* userData  = " + ntfyObj.getUserData());
     System.out.println("************************************************");
}
```

Administrative client program example

The following example is a complete administrative client program. Copy the contents to a file named `MyAdminClient.java`. After changing the node name and server name to the appropriate values for your configuration, you can compile and run it using the instructions from Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs

```
import java.util.Date;
import java.util.Properties;
import java.util.Set;

import javax.management.InstanceNotFoundException;
import javax.management.MalformedObjectNameException;
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.ObjectName;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.exception.ConnectorException;

public class AdminClientExample implements NotificationListener
{

    private AdminClient adminClient;
    private ObjectName nodeAgent;
    private long ntfyCount = 0;

    public static void main(String[] args)
    {
       AdminClientExample ace = new AdminClientExample();

       // Create an AdminClient
       ace.createAdminClient();

       // Find a NodeAgent MBean
       ace.getNodeAgentMBean("ellington");

       // Invoke launchProcess
       ace.invokeLaunchProcess("server1");

       // Register for NodeAgent events
       ace.registerNotificationListener();

       // Run until interrupted
       ace.countNotifications();
```

```
}

private void createAdminClient()
{
    // 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, "8879");

    // 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");
}


private void getNodeAgentMBean(String nodeName)
{
    // Query for the ObjectName of the NodeAgent MBean on the given node
    try
    {
        String query = "WebSphere:type=NodeAgent,node=" + nodeName + ",*";
        ObjectName queryName = new ObjectName(query);
        Set s = adminClient.queryNames(queryName, null);
        if (!s.isEmpty())
            nodeAgent = (ObjectName)s.iterator().next();
        else
        {
            System.out.println("Node agent MBean was not found");
            System.exit(-1);
        }
    }
    catch (MalformedObjectNameException e)
    {
        System.out.println(e);
        System.exit(-1);
    }
    catch (ConnectorException e)
    {
        System.out.println(e);
        System.exit(-1);
    }

    System.out.println("Found NodeAgent MBean for node " + nodeName);
}

private void invokeLaunchProcess(String serverName)
{
    // Use the launchProcess operation on the NodeAgent MBean to start
    // the given server
    String opName = "launchProcess";
    String signature[] = { "java.lang.String" };
    String params[] = { serverName };
    boolean launched = false;
    try
    {
        Boolean b = (Boolean)adminClient.invoke(
```

```
nodeAgent, opName, params, signature);
            launched = b.booleanValue();
            if (launched)
                System.out.println(serverName + " was launched");
            else
                System.out.println(serverName + " was not launched");

        }
        catch (Exception e)
        {
            System.out.println("Exception invoking launchProcess: " + e);
        }
    }

    private void registerNotificationListener()
    {
        // Register this object as a listener for notifications from the
        // NodeAgent MBean.  Don't use a filter and don't use a handback
        // object.
        try
        {
            adminClient.addNotificationListener(nodeAgent, this, null, null);
            System.out.println("Registered for event notifications");
        }
        catch (InstanceNotFoundException e)
        {
            System.out.println(e);
        }
        catch (ConnectorException e)
        {
            System.out.println(e);
        }
    }

    public void handleNotification(Notification ntfyObj, Object handback)
    {
        // Each notification that the NodeAgent MBean generates will result in
        // this method being called
        ntfyCount++;
        System.out.println("************************************************");
        System.out.println("* Notification received at " + new Date().toString());
        System.out.println("* type      = " + ntfyObj.getType());
        System.out.println("* message   = " + ntfyObj.getMessage());
        System.out.println("* source    = " + ntfyObj.getSource());
        System.out.println(
        "* seqNum    = " + Long.toString(ntfyObj.getSequenceNumber()));
        System.out.println("* timeStamp = " + new Date(ntfyObj.getTimeStamp()));
        System.out.println("* userData  = " + ntfyObj.getUserData());
        System.out.println("************************************************");

    }

    private void countNotifications()
    {
        // Run until killed
        try
        {
            while (true)
            {
                Thread.currentThread().sleep(60000);
                System.out.println(ntfyCount + " notification have been received");
            }
        }
        catch (InterruptedException e)
        {
```

```
        }
    }

}
```

Extending the WebSphere Application Server administrative system with custom MBeans

You can extend the WebSphere Application Server administration system by supplying and registering new Java Management Extensions (JMX) MBeans (see JMX 1.0 Specification for details) in one of the WebSphere processes. JMX MBeans represent the management interface for a particular piece of logic. All of the managed resources within the standard WebSphere infrastructure are represented as JMX MBeans. There are a variety of ways in which you can create your own MBeans and register them with the JMX MBeanServer running in any WebSphere process. For more information, view the MBean API documentation.

1. Create custom JMX MBeans.

   You have some alternatives to select from, when creating MBeans to extend the WebSphere administrative system. You can use any existing JMX MBean from another application. You can register any MBean that you tested in a JMX MBean server outside of the WebSphere Application Server environment in a WebSphere Application Server process, including standard MBeans, dynamic MBeans, open MBeans, and model MBeans.

   In addition to any existing JMX MBeans, and ones that were written and tested outside of the WebSphere Application Server environment, you can use the special distributed extensions provided by WebSphere and create a WebSphere ExtensionMBean provider. This alternative provides better integration with all of the distributed functions of the WebSphere administrative system. An ExtensionMBean provider implies that you supply an XML file that contains an MBean Descriptor based on the DTD shipped with the WebSphere Application Server. This descriptor tells the WebSphere system all of the attributes, operations, and notifications that your MBean supports. With this information, the WebSphere system can route remote requests to your MBean and register remote Listeners to receive your MBean event notifications.

   All of the internal WebSphere MBeans follow the Model MBean pattern (see WebSphere Application Server administrative MBean documentation for details). Pure Java classes supply the real logic for management functions, and the WebSphere MBeanFactory class reads the description of these functions from the XML MBean Descriptor and creates an instance of a ModelMBean that matches the descriptor. This ModelMBean instance is bound to your Java classes and registered with the MBeanServer running in the same process as your classes. Your Java code now becomes callable from any WebSphere Application Server administrative client through the ModelMBean created and registered to represent it.

2. Register the new MBeans. There are various ways to register your MBean.

   You can register your MBean with the WebSphere Application Server administrative service.

   You can register your MBean with the MBeanServer in a WebSphere Application Server process. The following list describes the available options in order of preference:

   - Go through the MBeanFactory class. If you want the greatest possible integration with the WebSphere Application Server system, then use the MBeanFactory class to manage the life cycle of your MBean through the activateMBean and deactivateMBean methods of the MBeanFactory class. Use these methods, by supplying a subclass of the RuntimeCollaborator abstract superclass and an XML MBean descriptor file. Using this approach, you supply a pure Java class that implements the management interface defined in the MBean descriptor. The MBeanFactory class creates the actual ModelMBean and registers it with the WebSphere Application Server administrative system on your behalf.

     This option is recommended for registering model MBeans.
   - Use the JMXManageable and CustomService interface. You can make the process of integrating with WebSphere administration even easier, by implementing a CustomService interface, that also

implements the JMXManageable interface. Using this approach, you can avoid supplying the RuntimeCollaborator. When your CustomService interface is initialized, the WebSphere MBeanFactory class reads your XML MBean descriptor file and creates, binds, and registers an MBean to your CustomService interface automatically. After the shutdown method of your CustomService is called, the WebSphere Application Server system automatically deactivates your MBean.

- Go through the AdminService interface. You can call the registerMBean() method on the AdminService interface and the invocation is delegated to the underlying MBeanServer for the process, after appropriate security checks. You can obtain a reference to the AdminService using the getAdminService() method of the AdminServiceFactory class.

  This option is recommended for registering standard, dynamic, and open MBeans. Implement the UserCollaborator class to use the MBeans and to provide a consistent level of support for them across distributed and z/OS platforms.
- Get MBeanServer instances directly. You can get a direct reference to the JMX MBeanServer instance running in any WebSphere Application Server process, by calling the getMBeanServer() method of the MBeanFactory class. You get a reference to the MBeanFactory class by calling the getMBeanFactory() method of the AdminService interface. Registering the MBean directly with the MBeanServer instance can result in that MBean not participating fully in the distributed features of the WebSphere Application Server administrative system.

Regardless of the approach used to create and register your MBean, you must set up proper Java 2 security permissions for your new MBean code. The WebSphere AdminService and MBeanServer are tightly protected using Java 2 security permissions and if you do not explicitly grant your code base permissions, security exceptions are thrown when you attempt to invoke methods of these classes. If you are supplying your MBean as part of your application, you can set the permissions in the `was.policy` file that you supply as part of your application metadata. If you are using a CustomService interface or other code that is not delivered as an application, you can edit the `library.policy` file in the node configuration, or even the `server.policy` file in the `properties` directory for a specific installation.

Best practices for standard, dynamic, and open MBeans

This article discusses recommended guidelines for standard, dynamic, and open MBeans.

The underlying interface for the WebSphere Application Server administrative service is AdminService. Remote access occurs through the AdminControl scripting object.

For WebSphere Application Server Version 5, the MBean registration and capabilities are as follows:

MBean type	Registered with:	Capabilities
Model	WebSphere Application Server administrative service	Local access is through the WebSphere Application Server administrative service or the MBean server. Remote access is through the WebSphere Application Server administrative service, and WebSphere Application Server security.
Standard, dynamic, or open	MBean server	Local access is through the WebSphere Application Server administrative service or the MBean server on the distributed platform.

For V6, you can optionally register standard, dynamic, and open custom MBeans with the WebSphere Application Server administrative service to take advantage of the capabilities that in V5 are available only to model MBeans.

V6 introduces a special run-time collaborator that you use with standard, dynamic or open custom MBeans to register the custom MBeans with the WebSphere Application Server administrative service. The standard, dynamic, and open MBeans display in the administrative service as model MBeans. The administrative service uses the capabilities available to MBeans that are registered with the administrative service.

For WebSphere Application Server Version 6, the MBean registration and capabilities are as follows:

MBean type	Registered with:	Capabilities
Model, and optionally standard, dynamic, or open	WebSphere Application Server administrative service	Local access is through the WebSphere Application Server administrative service or the MBean server. Remote access is through the WebSphere Application Server administrative service, and WebSphere Application Server security.
Standard, dynamic, or open	MBean server	Local access is through the WebSphere Application Server administrative service or the MBean server on the distributed platform.

Creating and registering standard, dynamic, and open custom MBeans

You can create standard, dynamic, and open custom MBeans and register them with the WebSphere Application Server administrative service.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Perform the following tasks to create and register a standard, dynamic, or open custom MBean.

1. Create your particular MBean class or classes.
2. Write an MBean descriptor in the XML language for your MBean.
3. Register your MBean by inserting code that uses the WebSphere Application Server run-time com.ibm.websphere.management.UserMBeanCollaborator collaborator class into your application code.
4. Package the class files for your MBean interface and implementation, the descriptor XML file, and your application Java archive (JAR) file.

After you successfully complete the steps, you have a standard, dynamic, or open custom MBean that is registered and activated with the WebSphere Application Server administrative service.

The following example shows how to create and register a standard MBean with the WebSphere Application Server administrative service:

```
SnoopMBean.java:

/**
 * Use the SnoopMBean MBean, which has a standard mbean interface.
 */
public interface SnoopMBean {
    public String getIdentification();
    public void snoopy(String parm1);
}

SnoopMBeanImpl.java:

/**
 * SnoopMBeanImpl - SnoopMBean implementation
```

```
 */
public class SnoopMBeanImpl implements SnoopMBean {
    public String getIdentification() {
        System.out.println(">>> getIdentification() called...");
        return "snoopy!";
    }

    public void snoopy(String parm1) {
        System.out.println(">>> snoopy(" + parm1 + ") called...");
    }
}
```

Define the following MBean descriptor for your MBean in an .xml file. The getIdentification method is set to run with the unicall option and the snoopy method is set to use the multicall option. These options are used only for z/OS platform applications. The WebSphere Application Server for z/OS options are not applicable to the distributed platforms, but they do not need to be removed. The options are ignored on the distributed platforms. Some statements are split on multiple lines for printing purposes.

SnoopMBean.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SnoopMBean"
 version="5.0"
 platform="dynamicproxy"
 description="Sample SnoopMBean to be initialized inside an EJB.">


 <attribute name="identification" getMethod="getIdentification" type="java.lang.String" proxyInvokeType="unicall"/>

 <operation name="snoopy" role="operation"  type="void" targetObjectType="objectReference"
impact="ACTION" proxyInvokeType="multicall">
  <signature>
   <parameter name="parm1" description="test parameter" type="java.lang.String"/>
  </signature>
 </operation>
</MBean>
```

Assume that your MBean is used in an enterprise bean. Register your MBean in the enterprise bean ejbCreate method and unregister it in the ejbRemove method.

```
//The method MBeanFactory.activateMBean() requires four parameters:
//String type: The type value that you put in this MBean's descriptor. For this example
//the string type is SnoopMBean.
//RuntimeCollaborator co: The UserMBeanCollaborator user MBean collaborator instance
//that you create
//String id: Unique name that you pick
//String desciptor: The MBean descriptor file name


import com.ibm.websphere.management.UserMBeanCollaborator;
//Import other classes here.
.
.
.
static private ObjectName snoopyON = null;
static private Object lockObj = "this is a lock";
.
.
.
/**
 * ejbCreate method: Register your Mbean.
 */
public void ejbCreate() throws javax.ejb.CreateException {
    synchronized (lockObj) {
        System.out.println(">>> SnoopMBean activating for --|" + this + "|--");
```

```
        if (snoopyON != null) {
            return;
        }
        try {
            System.out.println(">>> SnoopMBean activating...");
            MBeanFactory mbfactory = AdminServiceFactory.getMBeanFactory();
            RuntimeCollaborator snoop = new UserMBeanCollaborator(new SnoopMBeanImpl());
            snoopyON = mbfactory.activateMBean("SnoopMBean", snoop, "snoopMBeanId", "SnoopMBean.xml");
            System.out.println(">>> SnoopMBean activation COMPLETED! --|" + snoopyON + "|--");
        } catch (Exception e) {
            System.out.println(">>> SnoopMBean activation FAILED:");
            e.printStackTrace();
        }
    }
}
.
.
.
/**
 * ejbRemove method: Unregister your MBean.
 */
public void ejbRemove() {
    synchronized (lockObj) {
        System.out.println(">>> SnoopMBean Deactivating for --|" + this + "|--");
        if (snoopyON == null) {
            return;
        }
        try {
            System.out.println(">>> SnoopMBean Deactivating ==|" + snoopyON + "|== for --|" + this + "|--");
            MBeanFactory mbfactory = AdminServiceFactory.getMBeanFactory();
            mbfactory.deactivateMBean(snoopyON);
            System.out.println(">>> SnoopMBean Deactivation COMPLETED!");
        } catch (Exception e) {
            System.out.println(">>> SnoopMBean Deactivation FAILED:");
            e.printStackTrace();
        }
    }
}
```

Compile the MBean Java files and package the resulting class files with the descriptor .xml file, into the enterprise bean JAR file.

Java 2 security permissions

When you enable Java 2 security, you must grant Java 2 security permissions to application-specific code for Java Management Extensions (JMX) and WebSphere Application Server administrative privileges. With these permissions, your application code can call WebSphere Application Server administrative methods and JMX methods.

Use the following permission to invoke all the JMX class methods and interface methods:

```
permission javax.management.MBeanPermission "*", "*";
```

Consult the Java Management Extensions (JMX) API documentation for specific actions under the MBeanPermission class.

Use the following permission for WebSphere Application Server administrative application programming interfaces (APIs):

```
permission com.ibm.websphere.security.WebSphereRuntimePermission "AdminPermission";
```

Developing administrative programs for multiple Java 2 Platform, Enterprise Edition application servers

You can develop an administrative client to manage multiple vendor application servers through existing MBean support in the WebSphere Application Server.

Existence of MBeans for stopped components

The WebSphere Application Server completely implements the Java 2 Platform, Enterprise Edition (J2EE) Management specification. However, some differences in details between the J2EE specification and the WebSphere Application Server implementation are important for you to understand when you access WebSphere Application Server components. These differences are important to you when you access application MBeans because you can use either the WebSphere Application Server programming model or the J2EE programming model.

In the WebSphere Application Server programming model, if an MBean exists, you can assume that it is running. If an MBean does not exist, you can assume that it is stopped. Transient states between the started state and the stopped state are the same as the stopped state, which means that no MBean exists.

In the J2EE programming model, the MBean always exists regardless of the state of the component.

You can determine the state of a component by querying the state attribute. However, the state attribute only exists for MBeans that are state manageable, meaning that they implement the StateManageable interface. State manageable MBeans have start(), startRecursive(), and stop() operations whether these MBeans are J2EE MBeans or WebSphere Application Server MBeans. Additionally, the WebSphere Application Server defines the stateful interface. The stateful interface means that the component has a state and emits the j2ee.state.notifications method, but that the component cannot directly manage the state. For example, a Web module cannot stop itself. However, the application that contains the Web module can stop it.

Not all MBeans that have a state are state-manageable. Servlets, J2EE modules and enterprise beans, for example, are all stateful, but are not state manageable. The J2EE server is not state-manageable because no start() operation is available on a server.

The J2EEApplication MBean is an example of a state manageable MBean. When the WebSphere Application Server starts, each application activates a J2EEApplication MBean for itself. A J2EEApplication MBean has a J2EE type of J2EEApplication (for example, `ObjectName *:*,j2eeType=J2EEApplication`). If the application starts, it also activates an Application MBean with a type of Application (for example, `*:*,type=Application`). When the application changes state, the Application MBean is activated or deactivated. However, the J2EEApplication MBean is always activated. You can retrieve the application state changes by getting the state attribute.

The modules attribute on the J2EEApplication component returns an array of object names, one for every module in the application. The Application Server activates an MBean for each of these modules only after the Application Server starts the application. The managed enterprise bean isRegistered(ObjectName) method returns `false` if the application, and therefore the module, is not running.

All of the attributes that are defined in the J2EE management specification return valid values when the managed object stops. Other attributes and operations, for example those that are specifically defined for the Application Server, use the com.ibm.websphere.management.exception.ObjectNotRunningException exception if they are accessed when the object is stopped.

If you install the application while the server runs, the application installs the J2EEApplication MBean when the installation completes. Conversely, when the application uninstalls the J2EEApplication MBean, the application deactivates the MBean.

Mapping key properties

The following table lists the mapping from the J2EE management-defined j2eeType key property to the WebSphere Application Server type key property. You can use either key property to access MBeans. However, only use the j2eeType key property if you want to connect to application servers other than WebSphere Application Server.

j2eeType key property	type key property
J2EEDomain	J2EEDomain [new]
J2EEServer	Server
JVM	JVM
J2EEApplication	Application [separate MBean from j2eeType
WebModule	WebModule
ResourceAdapterModule	ResourceAdapterModule
EJBModule	EJBModule
EJB and subtypes / Servlet / ResourceAdapter	EJB and subtypes / Servlet / ResourceAdapter
JavaMailResource	MailProvider
JNDIResource	NameServer
JMSResource	JMSProvider
JTAResource	TransactionService
RMI_IIOPResource	ORB
URLResource	URLProvider
JDBCResource	JDBCProvider
JDBCDataSource	DataSource
JDBCDriver	JDBCDriver [new]
JCAResource	J2CResourceAdapter
JCAConnectionFactory	J2CConnectionFactory
JCAManagedConnectionFactory	J2CManagedConnectionFactory [new]

Optional WebSphere Application Server interfaces

The following table shows the optional J2EE management interfaces that WebSphere Application Server provides. Some j2eeType key properties are split on multiples lines for printing purproses.

j2eeType key property	EventProvider interface	StateManageable interface	StatisticsProvider interface
J2EEDomain	No	No	No
J2EEServer	Yes	Stateful	No
JVM	No	No	Yes
J2EEApplication	Yes	Yes	No
WebModule	Yes	Stateful	No
ResourceAdapterModule	Yes	Stateful	No

EJBModule	Yes	Stateful	No
AppClientModule	Yes	Stateful	No
EJB and subtypes / Servlet / ResourceAdapter	No	No	Yes
JavaMailResource	No	No	No
JNDIResource	No	No	No
JMSResource	No	No	No
JTAResource	Yes	No	Yes
RMI_IIOPResource	No	No	Yes
URLResource	No	No	No
JDBCResource	No	No	Yes
JDBCDataSource	No	No	No
JDBCDriver	No	No	No
JCAResource	Yes	No	Yes
JCAConnectionFactory	No	No	No
JCAManagedConnection Factory	No	No	No

Deploying and managing a custom Java administrative client program with multiple Java 2 Platform, Enterprise Edition application servers

This section describes how to connect to a Java 2 Platform, Enterprise Edition (J2EE) server, and how to manage multiple vendor servers.

The WebSphere Application Server completely implements the J2EE Management specification, also known as JSR-77 (Java Specification Requests 77). However, some differences in details between the J2EE specification and the WebSphere Application Server implementation are important for you to understand when you develop a Java administrative client program to manage multiple vendor servers. For information, see the Java Platform, Enterprise Edition (J2EE) Management Specification and the MBean application programming interface (API) documentation.

When your administrative client program accesses WebSphere Application Servers exclusively, you can use the Java APIs and WebSphere Application Server-defined MBeans to manage them. If your program needs to access both WebSphere Application Servers and other J2EE servers, use the API defined in the J2EE Management specification.

1. Connect to a J2EE server.

   Connect to a server by looking up the Management enterprise bean from the Java Naming and Directory Interface (JNDI). The Management enterprise bean supplies a remote interface to the MBean server that runs in the application server. The Management enterprise bean works almost exactly like the WebSphere Application Server administrative client, except that it does not provide WebSphere Application Server specific functionality. The following example shows how to look up the Management enterprise bean.

   ```
   import javax.management.j2ee.ManagementHome;
   import javax.management.j2ee.Management;

   Properties props = new Properties();

   props.setProperty(Context.PROVIDER_URL, "iiop://myhost:2809");
   Context ic = new InitialContext(props);
   ```

```
Object obj = ic.lookup("ejb/mgmt/MEJB");
ManagementHome mejbHome = (ManagementHome)
        PortableRemoteObject.narrow(obj, ManagementHome.class);
Management mejb = mejbHome.create();
```

The example gets an initial context to an application server by passing the host and port of the Remote Method Invocation (RMI) connector. You must explicitly code the RMI port, in this case 2809. The lookup method looks up the `ejb/mgmt/MEJB` path, which is the location of the Management enterprise bean home. The example then creates the mejb stateless session bean, which you use in the next step.

2. Manage multiple vendor application servers.

   After you create the mejb stateless session bean, you can use it to manage your application servers. Components from the application servers appear as MBeans, which the specification defines. These MBeans all have the j2eeType key property. This key property is one of a set of types that the specification defines. All of these types have a set of exposed attributes.

   Use the following example to guide you in managing multiple vendor application servers. The example uses the Java virtual machine (JVM) MBean to determine what the current heap size is for the application server.

```
ObjectName jvmQuery = new ObjectName("*:j2eeType=JVM,*");
Set s = mejb.queryNames(jvmQuery, null);
ObjectName jvmMBean = (ObjectName) s.iterator().next();
boolean hasStats = ((Boolean) mejb.getAttribute(jvmMBean,
        "statisticsProvider")).booleanValue();
if (hasStats) {
    JVMStats stats = (JVMStats) mejb.getAttribute(jvmMBean,
                                                  "stats");
    String[] statisticNames = stats.getStatisticNames();
    if (Arrays.asList(statisticNames).contains("heapSize")) {
        System.out.println("Heap size: " + stats.getHeapSize());
    }
}
```

   The queryNames() method first queries the JVM MBean. The getAttribute method gets the statisticsProvider attribute and determine if this MBean provides statistics. If the MBean does, the example accesses the stats attribute, and then invokes the getHeapSize() method to get the heap size.

The strength of this example is that the example can run on any vendor application server. It demonstrates that an MBean can optionally implement defined interfaces, in this case the StatisticsProvider interface. If an MBean implements the StatisticsProvider interface, you can see if an application server supports a particular statistic, in this case the heap size. The specification defines the heap size, although this value is optional. If the application server supports the heap size, you can display the heap size for the JVM.

Migrating Java Management Extensions V1.0 to Java Management Extensions V1.2

Each Java Virtual Machine (JVM) in WebSphere Application Server includes an embedded implementation of Java Management Extensions (JMX). In Application Server, Version 5, the JVMs contain an implementation of the JMX 1.0 specification. In Application Server, Version 6, the JVMs contain an implementation of the JMX 1.2 specification. The JMX 1.0 implementation used in Version 5 is the TMX4J package that IBM Tivoli products supply. The JMX 1.2 specification used in Version 6 is the open source mx4j package. The JMX implementation change across the releases does not affect the behavior of the JMX MBeans in the Application Server. No Application Server administrative application programming interfaces (APIs) are altered due to the change from the JMX V1.0 specification to the JMX V1.2 specification.

The JMX V1.2 specification is backward compatible with the JMX 1.0 specification. However, you might need to migrate custom MBeans that are supplied by products other than the Application Server from Version 5 to Version 6. The primary concern for these custom MBeans is related to the values that are used in key properties of the JMX ObjectName class for the MBean. The open source mx4j

implementation more stringently enforces property validation according to the JMX 1.2 specification. Test the custom MBeans that you deployed in Version 5 in Version 6, to ensure compatibility. Full details of the JMX V1.2 specification changes from the JMX V1.0 specification are available in the JMX 1.2 specification.

Java Management Extensions interoperability

WebSphere Application Server Version 6 implements Java Management Extensions (JMX) Version 1.2, while WebSphere Application Server Version 5 implements JMX Version 1.0.

Due to the evolution of the JMX specification, the serialization format for JMX objects, such as the javax.management.ObjectName object, differs between the V5 implementation and the V6 implementation. The V6 JMX run time is enhanced to be aware of the version of the client with which it is communicating. The V6 run time makes appropriate transformations on these incompatible serialized formats to support communication between the different version run times. A V5 wsadmin script or a V5 administrative client can call a V6 deployment manager, node, or server. A V6 wsadmin script or a V6 administrative client can call a V5 node or server.

When a V5 wsadmin script or a V5 administrative client calls a V6 MBean, the instances of classes that are new in V6 cannot be passed back to V5 because these classes are not present in the V5 environment. The problem occurs infrequently. However, it usually occurs when an exception embeds a nested exception that is new in V6. The symptom is usually a serialization exception or a NoClassDefFoundException exception.

Due to changes in the JMX implementation from V5 to V6, different exceptions are created when a method on an MBean is invoked for V5 than when a method on an MBean is invoked for V6. For example, when a method gets or sets an unknown attribute for V5, the MBeanRuntimeException exception is created. When a method gets or sets an unknown attribute for V6, the MBeanException exception that wraps a ServiceNotFoundException exception is created.

An instance of a user-defined class that implements the Serializable interface that is passed as a parameter or return value during MBean invocation, or sent as part of a notification, cannot contain a non-transient instance variable that is in the javax.management.package package. If the instance does, it cannot be properly deserialized when passed between V5 and V6 run times.

Due to changes in the supported format for the ObjectName class from V5 to V6, the configuration ID in V6 contains a vertical bar (|), whereas in V5, the ID contains a colon (:). This change is reflected in the output for wsadmin clients. For example, for a V5 client, the output is:

```
wsadmin>  $AdminConfig list Cell
     DefaultCellNetwork(cells/DefaultCellNetwork:cell.xml#Cell_1)
```

whereas for a V6 client, the output is:

```
wsadmin>  $AdminConfig list Cell
     DefaultCellNetwork(cells/DefaultCellNetwork|cell.xml#Cell_1)
```

The change to the configuration ID generally is not a problem because configuration IDs are generated dynamically. When a V5 client passes a configuration ID that contains a colon, the JMX run time, for upward compatibility, automatically transforms the configuration ID that contains a colon into a configuration ID that contains a vertical bar. Similarly, a reverse transformation is performed for backward compatibility.

Do not save the configuration ID and then try to use it later. Only query the ID and use it.

Managed object metadata

Information about a node, such as operating system platform and product features, is maintained in the configuration repository in the form of properties. As product features are installed on a node, new property settings are added.

WebSphere Application Server system management uses the managed object metadata properties as follows:

- To display the node version in the administrative console
- To ensure that new configuration types or attributes are not created or set on older release nodes
- To ensure that new resource types are not created on old release notes
- To ensure that new applications are not installed on old release nodes because the old run time cannot support the new applications

Base properties

The following base property keys are defined for WebSphere Application Server:

com.ibm.websphere.baseProductVersion: The version of WebSphere Application Server that is installed.

com.ibm.websphere.nodeOperatingSystem: The operating system platform on which the node runs.

com.ibm.websphere.nodeSysplexName: The sysplex name on a z/OS operating system.

com.ibm.websphere.deployed.features: A list of features that extends a profile. An example of a feature is an administrative console plug-in.

Here are examples of metadata property values. The last item is split on multiple lines for printing purposes.

```
com.ibm.websphere.baseProductVersion=6.0.0.0
com.ibm.websphere.nodeOperatingSystem=os390
com.ibm.websphere.nodeSysplexName=PLEX1
com.ibm.websphere.deployed.features=
 com.ibm.ws.base_6.0.0.0,com.ibm.ws.j2ee_6.0.0.0,
 com.ibm.ws.uddi_6.0.0.0,com.ibm.ws.wsgateway_6.0.0.0
```

For detailed information on metadata properties, view the ManagedObjectMetadataHelper class in the Application Server API documentation.

Accessing managed object metadata properties

An administrator can query managed object metadata through the wsadmin tool or Application Server APIs. They can additionally be viewed on the Node Installation properties administrative console panel. This article provides details on the Application Server API method.

An accessor class is used to obtain the managed object metadata properties. An accessor instance is created through its factory. A helper class, which uses the accessor instance, makes it easy to query the base metadata properties. These classes are all part of the com.ibm.websphere.management.metadata package in the Application Server API documentation. The specific names of these classes are:

- com.ibm.websphere.management.metadata.ManagedObjectMetadataHelper
- com.ibm.websphere.management.metadata.ManagedObjectMetadataAccessor
- com.ibm.websphere.management.metadata.ManagedObjectMetadataAccessorFactory

Managing applications through programming

This topic describes how, through Java MBean programming, to install, update, and delete a Java 2 Platform, Enterprise Edition (J2EE) application on WebSphere Application Server.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can install or change an application on WebSphere Application Server, you must first create or update your application and assemble it using an assembly tool.

Besides installing, uninstalling, and updating applications through programming, you can additionally install, uninstall, and update J2EE applications through the administrative console or the wsadmin tool. All three ways provide identical updating capabilities.

1. Perform any or all of the following tasks to manage your J2EE applications through programming.
   a. Install an application.

      This article provides an example for initially installing an application on WebSphere Application Server.
   b. Uninstall an application.

      This article provides an example for uninstalling an application that resides on WebSphere Application Server.
   c. Update an application.

      This article provides an example for updating the installed application on WebSphere Application Server with a new application. When you completely update an application, the deployed application is uninstalled and the new enterprise archive (EAR) file is installed.
   d. Add to, update, or delete part of an application.

      This article provides an example that you can use to add to, update, or delete part of an application on WebSphere Application Server.
   e. Add a module.

      This article provides an example for adding a module to an application that resides on WebSphere Application Server.
   f. Update a module.

      This article provides an example for updating a module that resides on WebSphere Application Server. When you update a module, the deployed module is uninstalled and the updated module is installed.
   g. Delete a module.

      This article provides an example for deleting a module that resides on WebSphere Application Server. When you delete a module, the deployed module is uninstalled.
   h. Add a file.

      This article provides an example for adding a file to an application that resides on WebSphere Application Server.
   i. Update a file.

      This article provides an example for updating a file on WebSphere Application Server. When you update a file, the deployed file is uninstalled and the updated file is installed.
   j. Delete a file.

      This article provides an example for deleting a file on WebSphere Application Server. When you delete a file, the deployed file is uninstalled.
2. Save your changes to the master configuration repository.
3. Synchronize changes to the master configuration across the nodes for the changes to take effect.

If you have further application updates, you can do the updates through programming, the administrative console, or the wsadmin tool.

Installing an application through programming

You can install an application through the administrative console, the wsadmin tool, or programming. Use this example to install an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can install an application on WebSphere Application Server, you must first create or update your application and assemble it using an assembly tool.

Perform the following tasks to install an application through programming.

1. Populate the enterprise archive (EAR) file with WebSphere Application Server-specific binding information.
   a. Create the controller and populate the EAR file with appropriate options.
   b. Optionally run the default binding generator.
   c. Save and close the EAR file.
   d. Retrieve the saved options table that will be passed to the installApplication MBean (API).
2. Connect to WebSphere Application Server.
3. Create the application management proxy.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the installApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.

      Refer to the com.ibm.websphere.management.application.AppManagement class in the Application Server API documentation to understand various options that can be passed to the installApplication MBean API.
5. Create the notification filter for listening to installation events.
6. Add the listener.
7. Install the application.
8. Wait for some timeout so that the program does not end.
9. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
10. When the installation is done, remove the listener and quit.

After you successfully run the code, the application is installed.

The following example shows how to install an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class Install {

    public static void main (String [] args) {
```

```
       try {
  String earFile = "C:/test/test.ear";
  String appName = "MyApp";

// Preparation phase: Begin
// Through the preparation phase you populate the enterprise archive (EAR) file with
// WebSphere Application Server-specific binding information. For example, you can specify
// Java Naming and Directory Interface (JNDI) names for enterprise beans, or virtual hosts
// for Web modules, and so on.

// First, create the controller and populate the EAR file with the appropriate options.
  Hashtable prefs = new Hashtable();
  prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// You can optionally run the default binding generator by using the following options.
// Refer to Java documentation for the AppDeploymentController class to see all the
// options that you can set.
  Properties defaultBnd = new Properties();
  prefs.put (AppConstants.APPDEPL_DFLTBNDG, defaultBnd);
  defaultBnd.put (AppConstants.APPDEPL_DFLTBNDG_VHOST, "default_host");

// Create the controller.
  AppDeploymentController controller = AppDeploymentController
    .readArchive(earFile, prefs);
  AppDeploymentTask task = controller.getFirstTask();
  while (task != null)
  {
// Populate the task data.
   String[][] data = task.getTaskData();
// Manipulate task data which is a table of stringtask.
   setTaskData (data);
   task = controller.getNextTask();
  }
  controller.saveAndClose();

  Hashtable options = controller.getAppDeploymentSavedResults();
// The previous options table contains the module-to-server relationship if it was set by
// using tasks.
//Preparation phase: End

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

// Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

// If code for the preparation phase has been run, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it.
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);
```

```
//Create the notification filter for listening to installation events.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (AppConstants.NotificationType);


//Add the listener.
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.INSTALL);

// Install the application.
  proxy.installApplication (earFile, appName, options, null);
  System.out.println ("After install App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
```

```
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Once you install the application, you must explicitly start the application or restart the server.

Starting an application through programming

You can start an application through the administrative console, the wsadmin tool, or programming. Use this example to start an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can start an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to start an application through programming.

1. Connect the administrative client to WebSphere Application Server.
2. Create the application management proxy.
3. Call the startApplication method on the proxy by passing the application name and optionally the list of targets on which to start the application.

After you successfully run the code, the application is started.

The following example shows how to start an application following the previously listed steps. Some statements are split on multiple lines for printing purposes.

```
//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;
String appName = "myApp";
Hashtable prefs = new Hashtable();
// Use the AppManagement MBean to start and stop applications on all or some targets.
// The AppManagement MBean is on the deployment manager in the Network Deployment product
// or on server1 in WebSphere Application Server.
// Query and get the AppManagement MBean.
ObjectName on = new ObjectName ("WebSphere:type=AppManagement,process=dmgr,*");
Iterator iter = client.queryNames (on, null).iterator();
ObjectName appmgmtON = (ObjectName)iter.next();

//Start the application on all targets.
AppManagement proxy = AppManagementProxy.getJMXProxyForClient(client);
String started = proxy.startApplication(appName, prefs, null);
System.out.println("Application started on folloing servers: " + started);

//Start the application on some targets.
//String targets = "WebSphere:cell=cellname,node=nodename,
server=servername+WebSphere:cell=cellname,cluster=clusterName";
//String started1 = proxy.startApplication(appName, targets, prefs, null);
//System.out.println("Application started on following servers: " + started1)
```

Uninstalling an application through programming

You can uninstall an application through the administrative console, the wsadmin tool, or programming. Use this example to uninstall an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can uninstall an application on WebSphere Application Server, you must first install it.

Perform the following tasks to uninstall an application through programming.

1. Get a connection to WebSphere Application Server.
2. Get the application management proxy.
3. Create the notification filter for listening to uninstallation events.
4. Add the listener.
5. Uninstall the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the uninstallation is done, remove the listener and quit.

After you successfully run the code, the application is uninstalled.

The following example shows how to uninstall an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class Uninstall {

    public static void main (String [] args) {

        try {

// Get a connection to the server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Get the application management proxy.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());


  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (AppConstants.NotificationType);


  //Add the listener.
```

```
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.UNINSTALL);

  // Uninstall the application.
  proxy.uninstallApplication (appName, options, null);
  System.out.println ("After uninstall App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the unistallation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
```

```
                    System.exit (0);
        }
    }
}
```

Updating an application through programming

You can update an existing application through the administrative console, the wsadmin tool, or programming. Use this example to completely update an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to completely updaste an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Prepare the enterprise archive (EAR) file by populating it with binding information.
6. Update the application.
7. Wait for some timeout so that the program does not end.
8. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
9. When the update is done, remove the listener and quit.

After you successfully run the code, the application is updated.

The following example shows how to update an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class aa {

    public static void main (String [] args) {

        try {

  // Connect to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);
```

```
  // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  String fileContents = "C:/test/test.ear";

  // Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.INSTALL);

// Refer to the installation example to see how you can prepare the enterprise archive (EAR)
// file by populating it with binding information.
// If code for the preparation phase has started, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put ((AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_APP);

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);
// Update the application.
  proxy.updateApplication (   appName,
                null,
                fileContents,
                AppConstants.APPUPDATE_UPDATE,
                options,
                null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
```

```
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Adding to, updating, or deleting part of an application through programming

You can add to, update, or delete part of an existing application through the administrative console, the wsadmin tool, or programming. This example changes part of an application through programming. You can use this example whether you add to, update, or delete part of an existing application. Multiple changes to an application can be packaged in a single .zip file.

To learn about the structure of the .zip file, see updating applications through the administrative console.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add to, update, or delete part of an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to add to, update, or delete part of an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter.
4. Add the listener.
5. Partially change the existing application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the update is done, remove the listener and quit.

After you successfully run the code, you have changed the application.

The following example shows how to add to, update, or delete part of an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//partialApp specifies the location of the partial application.
//appName specifies the name of the application.

String partialApp = "C:/apps/partial.zip";
String appName = "MyApp";

//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

// Create the notification filter.
NotificationFilterSupport myFilter = new NotificationFilterSupport();
myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
//Add the listener.
NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);
//Partially change the existing application, MyApp.

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_PARTIALAPP);

proxy.updateApplication ( appName,
    null,
    partialApp,
    null,
    options,
    null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
```

```
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Preparing a module and adding it to an existing application through programming

You can add a module to an existing application through the administrative console, the wsadmin tool, or programming. Use this example to add a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a module to an application on WebSphere Application Server, you must install the application.

Perform the following tasks to add a module to an application through programming.

1. Create an application deployment controller instance to populate the module file with binding information.
2. Save the binding information in the module.
3. Get the installation options.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Add the module to the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.

12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

After you successfully run the code, the module is added to the application.

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
//target specifies the cell, node, and server on which the module is installed.

String moduleName = "C:/apps/foo,jar";
String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java aArchive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_ADD,
    preferences,
    null);
```

If the module that you add to the application lacks any bindings, add the bindings so that the module addition works. Collect and add the bindings by using the public APIs provided with WebSphere Application Server. Refer to Java documentation for the com.ibm.websphere.management.application.client.AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server specific-binding information.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Connect the administrative client, AdminClient, to WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Update the existing application, MyApp, by adding the module.
String appName = "MyApp";

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);
```

```
//Specify the target for the new module.
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);
proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_ADD,
    options,
    null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
```

```
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
            }
        }
}
```

Preparing and updating a module through programming

You can update a module for an existing application through the administrative console, the wsadmin tool, or programming. When you update a module, you replace the existing module with a new version. Use this example to update a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a module on WebSphere Application Server, you must first install the application.

Perform the following tasks to update a module through programming.

1. Create an application deployment controller instance to populate the Java archive file with binding information.
2. Save the binding information in the module.
3. Get the installation options.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Replace the module in the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.
12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

After you successfully run the code, the existing module is replaced with the new one.

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
//target specifies the cell, node, and server on which the module is installed.
//appName specifies the name of the application to update.
String moduleName = "C:/apps/foo,jar";
String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";
String appName = "MyApp";
```

```
//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

Vector tasks = proxy.getApplicationInfo (appName, new Hashtable(), null);

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java archive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_UPDATE,
    preferences,
    tasks);
```

If the module that you update for the application lacks any bindings, add the bindings so that the module update works. Collect and add the bindings by using the public APIs that are provided with WebSphere Application Server. Refer to Java documentation for the AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server-specific binding information.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);


//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Update the existing application by adding the module.

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Specify the target for the new module
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);

proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);
// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
```

```
}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Deleting a module through programming

You can delete a module from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a module from an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to delete a module through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.

3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the module.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the module is deleted, remove the listener and quit.

After you successfully run the code, the existing module is deleted from the application.

The following example shows how to delete a module from an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//moduleURI specifies a URI that gives the target location of the module.
//appName specifies the name of the application to update.
String moduleURI = "Increment.jar";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.

AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the module.
Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);

proxy.updateApplication ( appName,
    moduleURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;
```

```
    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Adding a file through programming

You can add a file to an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to add a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a file to an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to add a file to an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Add the file to the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is added to the application, remove the listener and quit.

After you successfully run the code, the file is added to the application.

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class FileAdd {

    public static void main (String [] args) {

        try {

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

 // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  String fileURI = "test.war/com/acme/abc.jsp";
  String fileContents = "C:/temp/abc.jsp";

  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);

  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

  // Update the application
  proxy.updateApplication (   appName,
                fileURI,
                fileContents,
                AppConstants.APPUPDATE_ADD,
                options,
                null);

// Wait; the installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(90000); // Wait so that the program does not end.

        }
        catch (Exception e) {
```

```
            e.printStackTrace();
        }

    }

}


// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Updating a file through programming

You can update a file for an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to update a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a file for an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to update a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Update the file in the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the installation is done, remove the listener and quit.

After you successfully run the code, the file is updated for the application.

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileContents specifies the name of the file that you add to the application.
//appName specifies the name of the application.
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.

String fileContents = "C:/apps/test.jsp";
String appName = "MyApp";
String fileURI = "SomeWebMod.war/com/foo/abc.jsp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
    fileContents,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Deleting a file through programming

You can delete a file from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a file from an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to delete a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the file from the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is deleted from the application, remove the listener and quit.

After you successfully run the code, the file is deleted from the application.

The following example shows how to delete a file based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.
//appName specifies the name of the application.

String fileURI = "Increment.jar/com/acme/Foo.class";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the file.
Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait for some timeout. The installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
```

```
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //Once the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Chapter 8. Using command line tools

There are several command line tools that you can use to start, stop, and monitor WebSphere server processes and nodes. These tools only work on local servers and nodes. They cannot operate on a remote server or node. To administer a remote server, you can use the wsadmin scripting program connected to the deployment manager for the cell in which the target server or node is configured. See Deploying and managing using scripting for more information about using the wsadmin scripting program. You can also use the V5 administrative console which runs in the deployment manager for the cell. For more information about using the administrative console, see Deploying and managing with the GUI.

All command line tools function relative to a particular profile. If you run a command from a *install_root*/WebSphere/AppServer/bin directory, the command will run within the default profile. If you want to specify a different profile, perform one of the following:

- Specify the -profileName option. The profile that you specify with this option will be used instead of the default profile. For example:
  1. Change to the *install_root*/WebSphere/AppServer/bin directory.
  2. Type the following command: `startServer` *server1* `-profileName` *AppServerProfile*

  In this example, the command will function inside the *AppServerProfile* profile.
- Run the command from the `bin` directory of a specific profile. For example:
  1. Change to the *install_root*/WebSphere/AppServer/profiles/*MyProfile*/bin directory.
  2. Type the following command: `startServer` *server1*

  In this example, the command will function inside the *MyProfile* profile.

For more information about using profiles, including how to obtain a list of profiles, see the wasprofile command article.

To use the command line tools, perform the following steps:

1. Open a system command prompt.
2. Change to the `bin` directory.
3. Run the command.

The command runs the requested function and displays the results on the screen. Refer to the command log file for additional information. When you use the -trace option for the command, the additional trace data is captured in the command log file. The directory location for the log files is under the default system log root directory, except for commands related to a specific server instance, in which case the log directory for that server is used. You can override the default location for the command log file using the -logfile option for the command.

Example: Security and the command line tools

If you want to enable WebSphere Application Server security, you need to provide the command line tools with authentication information. Without authentication information, the command line tools receive an `AccessDenied` exception when you attempt to use them with security enabled. There are multiple ways to provide authentication data:

- Most command line tools support a -username and -password option for providing basic authentication data. Specify the user ID and password for an administrative user. For example, you can use a member of the administrative console users with operator or administrator privileges, or the administrative user ID configured in the user registry. The following example demonstrates the **stopNode** command, which specifies command line parameters:

  ```
  stopNode -username adminuser -password adminpw
  ```

© Copyright IBM Corp. 2004

- You can place the authentication data in a properties file that the command line tools read. The default file for this data is the `sas.client.props` file in the `properties` directory for the WebSphere Application Server.

startServer command

The **startServer** command reads the configuration file for the specified application server and starts the server. Depending on the options you specify, you can launch a new Java virtual machine (JVM) API to run the server process, or write the launch command data to a file. For more information about where to run this command, see the Using command tools article.

If you are using the Windows platform and the you have the application server running as a Windows service, the **startServer** command will start the associated Windows service and it will be responsible for starting the application server.

Syntax

The command syntax is as follows:

```
startServer <server> [options]
```

where `server` is the name of the application server you want to start. This argument is required.

Parameters

The following options are available for the **startServer** command:

-nowait
Tells the **startServer** command not to wait for successful initialization of the launched server process.

-quiet
Suppresses the progress information that the **startServer** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information to the log file for debugging purposes.

-timeout <seconds>
Specifies the waiting time before server initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-script [<script fileName>] -background
Generates a launch script with the **startServer** command instead of launching the server process directly. The launch script name is an optional argument. If you do not supply the launch script name, the default script file name is `start_<server>` based on the *<server>* name passed as the first argument to the **startServer** command. The `-background` parameter is an optional parameter that specifies that the generated script will run in the background when you execute it.

-J <java_option>
Specifies options to pass through to the Java interpreter.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
startServer server1

startServer server1 -script (produces the start_server1.sh or .bat files)

startServer server1 -trace (produces the startserver.log file)
```

stopServer command

The **stopServer** command reads the configuration file for the specified server process. This command sends a Java Management Extensions (JMX) command to the server telling it to shut down. By default, the **stopServer** command does not return control to the command line until the server completes the shut down process. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopServer** command. For more information about where to run this command, see the Using command tools article.

If you are using the Windows platform and the you have the application server running as a Windows service, the **stopServer** command will start the associated Windows service and it will be responsible for starting the application server.

Syntax

The command syntax is as follows:

```
stopServer <server> [options]
```

where *server* is the name of the configuration directory of the server you want to stop. This argument is required.

Parameters

The following options are available for the **stopServer** command:

-nowait
Tells the **stopServer** command not to wait for successful shutdown of the server process.

-quiet
Suppresses the progress information that the **stopServer** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the time to wait for server shutdown before timing out and returning an error.

-statusport <portNumber>
Supports an administrator in setting the port number for server status callback.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are Simple Object Access Protocol (SOAP), or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the server Java Management Extensions (JMX) port to use explicitly, so that you can avoid reading the configuration files to obtain the information.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

Note: If you are running in a secure environment but have not provided a user ID and password, you will receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
stopServer server1

stopServer server1 -nowait

stopServer server1 -trace (produces the stopserver.log file)
```

startManager command

The **startManager** command reads the configuration file for the Network Deployment manager process and constructs a **launch** command. Depending on the options you specify, the **startManager** command launches a new Java virtual machine (JVM) API to run the manager process, or writes the **launch** command data to a file. You must run this command from the *install_root*/WebSphere/AppServer/profiles/standalone/bin directory of a Network Deployment installation.

If you are using the Windows platform and the you have the deployment manager running as a Windows service, the **startManager** command will start the associated Windows service and it will be responsible for starting the deployment manager.

Syntax

The command syntax is as follows:

```
startManager [options]
```

Parameters

The following options are available for the **startManager** command:

-nowait
Tells the **startManager** command not to wait for successful initialization of the deployment manager process.

-quiet
Suppresses the progress information that the **startManager** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file using the **startManager** command for debugging purposes.

-timeout <seconds>
Specifies the waiting time before deployment manager initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for deployment manager status callback.

-script [<script fileName>] -background
Generates a launch script with the **startManager** command instead of launching the deployment manager process directly. The launch script name is an optional argument. If you do not provide the launch script name, the default script file name is `<start_dmgr>`. The `-background` parameter is an optional parameter that specifies that the generated script will run in the background when you execute it.

-J-<java_option>
Specifies options to pass through to the Java interpreter.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
startManager

startManager -script (produces the start_dmgr.sh or .bat file)

startManager -trace (produces the startmanager.log file)
```

stopManager command

The **stopManager** command reads the configuration file for the Network Deployment manager process. It sends a Java Management Extensions (JMX) command to the manager telling it to shut down. By default, the **stopManager** command waits for the manager to complete the shutdown process before it returns control to the command line. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopManager** command. For more information about where to run this command, see the Using command tools article.

If you are using the Windows platform and the you have the deployment manager running as a Windows service, the **stopManager** command will start the associated Windows service and it will be responsible for starting the deployment manager.

Syntax

The command syntax is as follows:

```
stopManager [options]
```

Parameters

The following options are available for the **stopManager** command:

-nowait
Tells the **stopManager** command not to wait for successful shutdown of the deployment manager process.

-quiet
Suppresses the progress information that the **stopManager** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information to a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time for the manager to complete shutdown before timing out and returning an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are Simple Object Access Protocol (SOAP) or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the deployment manager JMX port to use explicitly, so that you can avoid reading the configuration files to obtain information.

-username <name>
Specifies the user name for authentication if security is enabled in the deployment manager. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the deployment manager. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the deployment manager.

Note: If you are running in a secure environment but have not provided a user ID and password, you receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
stopManager

stopManager -nowait

stopManager -trace (produces the stopmanager.log file)
```

startNode command

The **startNode** command reads the configuration file for the node agent process and constructs a **launch** command. Depending on the options that you specify, the **startNode** command creates a new Java virtual machine (JVM) API to run the agent process, or writes the launch command data to a file. For more information about where to run this command, see the Using command tools article.

If you are using the Windows platform and the you have the node agent running as a Windows service, the **startNode** command will start the associated Windows service and it will be responsible for starting the node agent.

Syntax

The command syntax is as follows:

```
startNode [options]
```

Parameters

The following options are available for the **startNode** command:

-nowait
Tells the **startNode** command not to wait for successful initialization of the node agent process.

-quiet
Suppresses the progress information that the **startNode** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time before node agent initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for node agent status callback.

-script [<script fileName>] -background
Generates a launch script with the **startNode** command instead of launching the node agent process directly. The launch script name is an optional argument. If you do not provide the launch script name, the default script file name is `start_<nodeName>`, based on the name of the node. The `-background` parameter is an optional parameter that specifies that the generated script will run in the background when you execute it.

-J-<java_option>
Specifies options to pass through to the Java interpreter.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
startNode

startNode -script (produces the start_node.bat or .sh file)

startNode -trace (produces the startnode.log file)
```

stopNode command

The **stopNode** command reads the configuration file for the Network Deployment node agent process and sends a Java Management Extensions (JMX) command telling the node agent to shut down. By default, the **stopNode** command waits for the node agent to complete shutdown before it returns control to the command line. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopNode** command. For more information about where to run this command, see the Using command tools article.

If you are using the Windows platform and the you have the node agent running as a Windows service, the **stopNode** command will start the associated Windows service and it will be responsible for starting the node agent.

Syntax

The command syntax is as follows:

```
stopNode [options]
```

Parameters

The following options are available for the **stopNode** command:

-nowait
Tells the **stopNode** command not to wait for successful shutdown of the node agent process.

-quiet
Suppresses the progress information that the **stopNode** command prints in normal mode.

-logfile <fileNname>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time for the agent to shut down before timing out and returning an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-stopservers
Stops all application servers on the node before stopping the node agent.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are Simple Object Access Protocol (SOAP) or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the node agent JMX port to use explicitly, so that you can avoid reading configuration files to obtain the information.

-username <name>
Specifies the user name for authentication if security is enabled in the node agent. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the node agent. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the node agent.

Note: If you are running in a secure environment but have not provided a user ID and password, you receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

Note: When requesting help for the usage statement for the **stopNode** command, a reference to the **stopServer** command displays. All of the options displayed for this usage statement apply to the **stopNode** command.

-? Prints a usage statement.

Note: When requesting help for the usage statement for the **stopNode** command, a reference to the **stopServer** command displays. All of the options displayed for this usage statement apply to the **stopNode** command.

Usage scenario

The following examples demonstrate correct syntax:

```
stopNode

stopNode -nowait

stopNode -trace (produces the stopnode.log file)
```

addNode command

The **addNode** command incorporates a WebSphere Application Server installation into a cell. For more information about where to run this command, see the Using command tools article. Depending on the size and location of the new node you incorporate into the cell, this command can take a few minutes to complete.

The node agent server is automatically started as part of the **addNode** command unless you specify the `-noagent` option. If you recycle the system that hosts an application server node, and did not set up the node agent to act as an operating system daemon, you must issue a **startNode** command to start the node agent before starting any application servers.

The following items are new in V6:

- Ports generated for the node agent are unique for all the profiles in the installation. For development purposes, you can create multiple profiles on the same installation and add them to one or more cells without worrying about ports conflicts.
- If you want to specify the ports that the node agent uses, specify it is a file with the file name passed with the -portprops option. The format of the file is key=value pairs, one on each line, with the key being the same as the port name in the serverindex.xml file.
- If you want to use a number of sequential ports, the -startingport option works the same as it does in V5.x. This means that port conflicts with other profiles will not be detected.

Syntax

The command syntax is as follows:

```
addNode dmgr_host [dmgr_port] [-conntype type] [-includeapps]
[-startingport portnumber] [-portprops qualified_filename]
[-nodeagentshortname name] [-nodegroupname name] [-includebuses name]
[-registerservice] [-servicename name] [-servicepassword password]
[-coregroupname name] [-noagent] [-statusport port] [-quiet] [-nowait]
[-logfile filename] [-replacelog] [-trace] [-username uid]
[-password pwd] [-help]
```

The `dmgr_host` argument is required. All of the other arguments are optional. The default port number is 8879 for the default Simple Object Access Protocol (SOAP) port of the deployment manager. SOAP is the default Java Management Extensions (JMX) connector type for the command. If you have multiple WebSphere Application Server installations or multiple profiles, the SOAP port may be different than 8879. Examine the deployment manager SystemOut.log to see the current ports in use.

Parameters

The following options are available for the **addNode** command:

-conntype <type>
Specifies the JMX connector type to use for connecting to the deployment manager. Valid types are SOAP or RMI, which stands for Remote Method Invocation.

-includeapps
By default the **addNode** command does not carry over applications from the stand-alone servers on the new node to the cell. In general, you should install applications using the deployment manager. The -includeapps option tells the **addNode** command to carry over the applications from a node. If the application already exists in the cell, a warning is printed and the application does not install in the cell.

The applications will be mapped to the server that you federated using the **addNode** command. When the **addNode** command operation completes, the applications will run on that server when the server is started. Since these applications are part of the network deployment cell, you can map them to other servers and clusters in the cell using the administrative console. See the Mapping modules to servers article for more information.

By default, during application installation, application binaries are extracted in the *install_root*/installedApps/cellName directory. After the **addNode** command, the cell name of the configuration on the node that you added changes from the base cell name to the deployment manager cell name. The application binaries are located where they were before the **addNode** command ran, for example, *install_root*/installedApps/old_cellName.

If the application was installed by explicitly specifying the location for binaries as the following example:

${*INSTALL_ROOT*}/${CELL}

where the variable ${CELL}, specifies the current cell name, then when the **addNode** command runs, the binaries are moved to the following directory:

${*INSTALL_ROOT*}/currentCellName

Federating the node to a cell using the **addNode** command does not merge any cell level configuration, including virtual host information. If the virtual host and aliases for the new cell do not match WebSphere Application Server, you cannot access the applications running on the servers. You have to manually add all the virtual host and host aliases to the new cell, using the administrative console running on the deployment manager.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-user <name> or -username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option. The user name that you choose must be a pre-existing user name.

-nowait
Tells the **addNode** command not to wait for successful initialization of the launched node agent process.

-quiet
Suppresses the progress information that the **addNode** command prints in normal mode.

-logfile <filename>
Specifies the location of the log file to which information gets written. By default, the log file is called addNode.log and is created in the logs directory of the profile for the node being added.

-trace
Generates additional trace information in the log file for debugging purposes.

-replacelog
Replaces the log file instead of appending to the current log. By default, the **addNode** command appends to the existing trace file. This option causes the **addNode** command to overwrite the trace file.

-noagent
Tells the **addNode** command not to launch the node agent process for the new node.

-password <password>
Specifies the password for authentication if security is enabled. The password that you choose must be one that is associated with a pre-existing user name.

-startingport <portNumber>
Supports the specification of a port number to use as the base port number for all node agent ports created during the **addNode** command. With this support you can control which ports are defined for these servers, rather than using the default port values. The starting port number is incremented to calculate the port number for every node agent port configured during the **addNode** command.

-registerservice
(Windows only) Registers the node agent as a Windows service.

-servicename <user>
(Windows only) Use the given user name as the Windows service user.

-servicepassword <password>
(Windows password) Use the given password as the Windows service password.

-portprops <filename>
Passes the name of the file that contains key-value pairs of explicit ports that you want the new node agent to use. For example, to set your SOAP and RMI ports to 3000 and 3001, create a file with the following two lines and pass it as the parameter:

```
SOAP_CONNECTOR_ADDRESS=3000
BOOTSTRAP_ADDRESS=3001
```

-coregroupname <name>
The name of the core group in which to add this node. If you do not specify this option, the node will be added to the DefaultCoreGroup.

-nodegroupname <name>
The name of the node group in which to add this node. If you do not specify, the node is added to the DefaultNodeGroup.

-includebuses
Copies the buses from the node to be federated to the cell.

-nodeagentshortname <name>
The shortname to use for the new node agent.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
addNode testhost 8879 (adds an Application Server to the deployment manager)

addNode deploymgr 8879 -trace (produces the addNode.log file)

addNode host25 8879 -nowait (does not wait for a node agent process)
```

where 8879 is the default port.

Best practices for adding nodes using command line tools

Use the **addNode** command to add a standalone node into a cell. The **addNode** command does the following:

- Copies the base WebSphere Application Server cell configuration to a new cell structure. This new cell structure matches the structure of deployment manager.
- Creates a new node agent definition for the node that the cell incorporates.
- Sends commands to the deployment manager to add the documents from the new node to the cell repository.
- Performs the first configuration synchronization for the new node, and verifies that this node is synchronized with the cell.
- Launches the node agent process for the new node.
- Updates the setupCmdLine.bat or setupCmdline.sh files and the wsadmin.properties file to point to the new cell.
- After federating the node, the **addNode** command backs up the `plugin-cfg.xml` file from the `<install_root>/config/cells` directory to the `config/backup/base/cells` directory. The **addNode** command regenerates a new `plugin-cfg.xml` file at the Deployment Manger and the nodeSync operation copies the files to the node level.

  For information about port numbers, see the article Port number settings in WebSphere Application Server versions.

Tips for using the **addNode** command:

- Do not put WebSphere Application Server Jar files on the generic `CLASSPATH` variable (default class path) for the overall system.
- **Unix/Linux users:** Some Unix or Linux systems create an association between the host name of the machine and the loopback address -- 127.0.0.1 (Red Hat installations do this by default). In addition, the `/etc/nsswitch.conf` file is set up to use the `/etc/hosts` path before trying to look up the server using a name server. This setup can cause failures when trying to add or administrate nodes when the deployment manager or application server is running on the Red Hat system or an Unix/Linux system with the same setup.

  If your deployment manager or your application server run on the Red Hat system, or an Unix/Linux system with the same setup, perform the following operations to ensure that you can successfully add and administer nodes:
  - Remove the 127.0.0.1 mapping to the local host in the `/etc/hosts` path.
  - Edit the `/etc/nsswitch.conf` file so that the hosts line reads:

    ```
    hosts:  dns files
    ```
- By default, applications that are installed on the node will not copy to the cell. If you install an application after using the **addNode** command, the application will install on the cell. By specifying the `-includeapps` option, you force the **addNode** command to copy applications from the node to the cell. Applications with duplicate names will not copy to the cell.
- Cell-level documents are not merged. Any changes that you make to the standalone cell-level documents before using the **addNode** command must be repeated on the new cell. For example, virtual hosts.

serverStatus command

Use the **serverStatus** command to obtain the status of one or all of the servers configured on a node. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
serverStatus <server>|-all [options]
```

The first argument is required. The argument is either the name of the server for which status is desired, or the -all keyword which requests status for all servers defined on the node.

Parameters

The following options are available for the **serverStatus** command:

-quiet
Suppresses the progress information that the **serverStatus** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
serverStatus server1

serverStatus -all (returns status for all defined servers)

serverStatus -trace (produces the serverStatus.log file)
```

removeNode command

The **removeNode** command returns a node from a Network Deployment distributed administration cell to a base WebSphere Application Server installation. For more information about where to run this command, see the Using command tools article.

The **removeNode** command only removes the node-specific configuration from the cell. This command does not uninstall any applications that were installed as the result of executing an **addNode** command. Such applications can subsequently deploy on additional servers in the Network Deployment cell. As a consequence, an **addNode** command with the -includeapps option executed after a **removeNode** command does not move the applications into the cell because they already exist from the first **addNode** command. The resulting application servers added on the node do not contain any applications. To deal with this situation, add the node and use the deployment manager to manage the applications. Add the applications to the servers on the node after it is incorporated into the cell.

The **removeNode** command does the following:
- Stops all of the running server processes in the node, including the node agent process.
- Removes the configuration documents for the node from the cell repository by sending commands to the deployment manager.
- Copies the original application server cell configuration into the active configuration.

Depending on the size and location of the new node you remove from the cell, this command can take a few minutes to complete.

Syntax

The command syntax is as follows:

```
removeNode [options]
```

All arguments are optional.

Parameters

The following options are available for the **removeNode** command:

-quiet
Suppresses the progress information that the **removeNode** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-statusport <portNumber>
Specifies that an administrator can set the port number for the node agent status callback.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-force
Cleans up the local node configuration regardless of whether you can reach the deployment manager for cell repository cleanup. After using the `-force` parameter, you may need to use the **cleanupNode** command on the deployment manager.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
removeNode -quiet

removeNode -trace (produces the removeNode.log file)
```

cleanupNode command

The **cleanupNode** command cleans up a node configuration from the cell repository. Only use this command to clean up a node if you have a node defined in the cell configuration, but the node no longer exists. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
cleanupNode <node name> <deploymgr host> <deploymgr port> [options]
```

where the first argument is required.

Parameters

The following options are available for the **cleanupNode** command:

-quiet
Suppresses the progress information that the **cleanupNode** command prints in normal mode.

-trace
Generates trace information into a file for debugging purposes.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

Usage scenario

The following examples demonstrate correct syntax:

```
cleanupNode myNode
cleanupNode myNode -trace
```

syncNode command

The **syncNode** command forces a configuration synchronization to occur between the node and the deployment manager for the cell in which the node is configured.

The node agent server runs a configuration synchronization service that keeps the node configuration synchronized with the master cell configuration. If the node agent is unable to run because of a problem in the node configuration, you can use the **syncNode** command to perform a synchronization when the deployment manager is not running in order to force the node configuration back in sync with the cell configuration.

For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
syncNode <deploymgr host> <deploymgr port> [options]
```

where the <deploymgr *host*> argument is required.

Parameters

The following options are available for the **syncNode** command:

-stopservers
Tells the **syncNode** command to stop all servers on the node, including the node agent, before performing configuration synchronization with the cell.

-restart
Tells the **syncNode** command to launch the node agent process after configuration synchronization completes.

-nowait
Tells the **syncNode** command not to wait for successful initialization of the launched node agent process.

-quiet
Suppresses the progress information that the **syncNode** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time before node agent initialization times out and returns an error.

-statusport <portnumber>
Specifies that an administrator can set the port number for node agent status callback.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are Simple Object Access Protocol (SOAP) or Remote Method Invocation (RMI).

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
syncNode testhost 8879

syncNode deploymgr 8879 -trace (produces the syncNode.log file)

syncNode host25 4444 -stopservers -restart
(assumes that the deployment manager JMX port is 4444)
```

backupConfig command

The **backupConfig** command is a simple utility to back up the configuration of your node to a file. By default, all servers on the node stop before the backup is made so that partially synchronized information is not saved. For more information about where to run this command, see the Using command tools article. If you do not have root authority, you must specify a path for the backup file in a location where you have write permission. The backup file will be in zip format and a `.zip` extension is recommended.

Syntax

The command syntax is as follows:

```
backupConfig <backup_file> [options]
```

where *backup_file* specifies the file to which the backup is written. If you do not specify one, a unique name is generated.

Parameters

The following options are available for the **backupConfig** command:

-nostop
Tells the **backupConfig** command not to stop the servers before backing up the configuration.

-quiet
Suppresses the progress information that the **backupConfig** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into the log file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following example demonstrates correct syntax:

```
backupConfig
```

This example creates a new file that includes the current date. For example: `WebSphereConfig_2003-04-22.zip`

```
backupConfig myBackup.zip -nostop
```

This example creates a file called `myBackup.zip`, and does not stop any servers before beginning the backup process.

restoreConfig command

The **restoreConfig** command is a simple utility to restore the configuration of your node after backing up the configuration using the **backupConfig** command. By default, all servers on the node stop before the configuration restores so that a node synchronization does not occur during the restoration. If the configuration directory already exists, it is renamed before the restoration occurs. For more information about where to run this command, see the Using command tools article.

For AIX only, if you are using a logical directory for `was_install/config`, the **restoreConfig** command will not work.

Syntax

The command syntax is as follows:

```
restoreConfig <backup_file> [options]
```

where *backup_file* specifies the file to be restored. If you do not specify one, the command will not run.

Parameters

The following options are available for the **restoreConfig** command:

-nowait
Tells the **restoreConfig** command not to stop the servers before restoring the configuration.

-quiet
Suppresses the progress information that the **restoreConfig** command prints in normal mode.

-location <directory_name>
Specifies the directory where the backup file is restored. The location defaults to the `install_root/config` directory.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into the log file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following example demonstrates correct syntax:

```
restoreConfig WebSphereConfig_2003-04-22.zip
```

The following example restores the given file to the `/tmp` directory and does not stop any servers before beginning the restoration:

```
restoreConfig WebSphereConfig_2003-04-22.zip -location /tmp -nostop
```

Be aware that if you restore the configuration to a directory that is different from the directory that was backed up when you performed the **backupConfig** command, you may need to manually update some of the paths in the configuration directory.

EARExpander command

Use the **EARExpander** command to expand an enterprise archive file (EAR) into a directory to run the application in that EAR file. You can collapse a directory containing application files into a single EAR file. You can type `EARExpander` with no arguments to learn more about its options. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
EarExpander -ear earName -operationDir dirName -operation
<expand | collapse> [-expansionFlags <all|war>]
```

Parameters

The following options are available for the **EARExpander** command:

-ear
Specifies the name of the input EAR file for the expand operation or the name of the output EAR file for the collapse operation.

-operationDir
Specifies the directory where the EAR file is expanded or specifies the directory from where files are collapsed.

-operation <expand | collapse>
The `expand` value expands an EAR file into a directory structure required by the WebSphere Application Server run time. The `collapse` value creates an EAR file from an expanded directory structure.

-expansionFlags <all | war>
(Optional) The `all` value expands all files from all of the modules. The `war` value only expands the files from Web archive file (WAR) modules.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

Usage scenario

The following examples demonstrate correct syntax:

```
EARExpander -ear C:\WebSphere\AppServer\installableApps\DefaultApplication.ear
-operationDir  C:\MyApps -operation expand -expansionFlags war

EARExpander -ear C:\backup\DefaultApplication.ear
-operationDir C:\MyAppsDefaultApplication.ear -operation collapse
```

GenPluginCfg command

This topic describes the command-line syntax for the GenPluginCfg command. This command is used to regenerate the WebSphere Web server plug-in configuration file, plugin-cfg.xml. For more information about where to run this command, see the Using command tools article.

CAUTION:
Regenerating the plug-in configuration can overwrite manual configuration changes that you might want to preserve. Before performing this task, understand its implications as described in the article Communicating with Web servers.

To regenerate the plug-in configuration, you can either click on **Servers > Web Servers** in the administrative console, select a Web server and then click Generate Plug-in, or you can issue the following command:

```
GenPluginCfg.sh|bat
```

Both methods for regenerating the plug-in configuration create a `plugin-cfg.xml` file in ASCII format, which is the proper format for execution in a distributed environment.

You can use the -profileName option to define the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

Syntax

The command syntax is as follows:

```
GenPluginCfg [[-option.name optionValue]...]
```

When the GenPluginCfg command is issued with the option -webserver.name webservrName, wsadmin generates a plug-in configuration file for the Web server. This settings in this generated configuration file are based on the list of applications that are deployed on the Web server. When this command is issued without the option -webserver.name webservrName, the plug-in configuration file is generated based on topology.

Parameters

The following options are available for the **startServer** command:

-config.root configroot_dir
Defaults to environment variable CONFIG_ROOT.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-cell.name cell
Defaults to environment variable WAS_CELL.

-node.name node
Defaults to environment variable WAS_NODE.

-webserver.name webserver1
Required for creating plug-in configuration file for a given Web server.

-propagate yes/no
Applicable only when the option webserver.name is specified. Defaults to no.

-cluster.name cluster1,cluster2 | ALL
Optional list of clusters. Ignored when the option webserver.name is specified.

-server.name server1,server2
Optional list of servers. Required for single server plug-in generation. Ignored when the option webserver.name is specified.

-output.file.name file_name
Defaults to the configroot_dir/plugin-cfg.xml file. Ignored when the option webserver.name is specified.

-destination.root root
Installation root of the machine configuration is used on. Ignored when the option webserver.name is specified.

-destination.operating.system windows/unix
Operating system of the machine configuration is used on. Ignored when the option webserver.name is specified.

-debug yes/no
Defaults to no.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

To generate a plug-in configuration for all of the clusters in a cell:

```
GenPluginCfg -cell.name NetworkDeploymentCell
```

To generate a plug-in configuration for a single server:

```
GenPluginCfg -cell.name BaseApplicationServerCell
-node.name appServerNode -server.name appServerName
```

To generate a plug-in configuration file for a Web server:

```
GenPluginCfg -cell.name BaseApplicationServerCell
-node.name webserverNode -webserver.name webserverName
```

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 Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

© Copyright IBM Corp. 2004

Trademarks and service marks

For trademark attribution, visit the IBM Terms of Use Web site (http://www.ibm.com/legal/us/).

© Copyright IBM Corp. 2004