



System Administration

Note

Before using this information, be sure to read the general information under “Notices” on page 339.

Compilation date: May 20, 2004

© Copyright International Business Machines Corporation 2003, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

How to send your comments vii

Chapter 1. Welcome to System Administration 1

Chapter 2. Welcome to the Administrative Console 3

Chapter 3. Identifying where to perform WebSphere Application Server operations. 5

Chapter 4. Using the administrative console 9

Starting and stopping the administrative console 9
 Login settings 10
Administrative console areas 11
 Taskbar 12
 Navigation tree 12
 WorkSpace 12
 WebSphere Status 12
 Administrative console buttons 12
 Administrative console page features 15
 Administrative console navigation tree actions. 16
 Administrative console taskbar actions 17
 WebSphere status settings 17
Specifying console preferences 18
 Preferences settings. 18
 Administrative console preference settings 19
 Administrative console filter settings 20
 Administrative console scope settings 20
Accessing help 20
Administrative console: Resources for learning 21

Chapter 5. Using the MVS console 23

Chapter 6. Welcome to Scripting 25

Chapter 7. Deploying and managing using scripting 27

Migrating from Systems Management Scripting API (SMAPI) V4 to wsadmin V5 27
 Example: Migrating - Starting an application server 29
 Example: Migrating - Installing an application. 30
 Example: Migrating - Uninstalling an application 32
 Example: Migrating - Defining a Java™ 2 Enterprise Edition resource 33
 Example: Migrating - Stopping an application server 34
Launching scripting clients 35
 Wsadmin tool 37
 Jacl 40

Scripting objects 41
 Help object for scripted administration 42
 AdminApp object for scripted administration 51
 AdminControl object for scripted administration 75
 AdminConfig object for scripted administration 93
 ObjectName, Attribute, and AttributeList 111
 Modifying nested attributes with the wsadmin tool 112
 Managing configurations with scripting 113
 Creating configuration objects using the wsadmin tool 114
 Specifying configuration objects using the wsadmin tool 116
 Listing attributes of configuration objects using the wsadmin tool 118
 Modifying configuration objects with the wsadmin tool 120
 Removing configuration objects with the wsadmin tool 122
 Changing the WebSphere Application Server configuration using wsadmin 123
 Configuration management examples with wsadmin 126
 Managing running objects with scripting 193
 Specifying running objects using the wsadmin tool. 193
 Identifying attributes and operations for running objects with the wsadmin tool 195
 Performing operations on running objects using the wsadmin tool 196
 Modifying attributes on running objects with the wsadmin tool 198
 Operation management examples with wsadmin 199
 Managing applications with scripting 215
 Installing applications with the wsadmin tool 215
 Installing stand-alone Java archive and Web archive files with the wsadmin tool 217
 Listing applications with the wsadmin tool 218
 Editing application configurations with the wsadmin tool 218
 Uninstalling applications with the wsadmin tool 219
 Application management examples with wsadmin 220
 wsadmin scripting environment 235
 wsadmin traces. 235
 Tracing operations with the wsadmin tool. 236
 Profiles and scripting. 237
 Properties used by scripted administration 237
 Java Management Extensions connectors 239
 Security and scripting 240
 Scripting management examples with wsadmin 241
 wsadmin tool performance tips 243

Chapter 8. Welcome to Commands 245

Chapter 9. Managing using command line tools 247

Example: Security and the command line tools	247
startServer command	247
stopServer command	248
startManager command	250
stopManager command	251
stopNode command	252
START <i>appserver_proc_name</i> command	254
STOP <i>appserver_proc_name</i> command	255
START <i>dmgr_proc_name</i> command	255
STOP <i>dmgr_proc_name</i> command	256
START <i>nodeagent_proc_name</i> command	256
STOP <i>nodeagent_proc_name</i> command	257
addNode command	258
Best practices for adding nodes using command line tools	260
serverStatus command	261
removeNode command	262
cleanupNode command	264
syncNode command	264
backupConfig command	266
restoreConfig command	267
EARExpander command	268
GenPluginCfg command	269

Chapter 10. Managing using mvs command line tools 271

Modify command	271
Example: Canceling application clusters and servers with the modify command	275
Example: Establishing a general level of trace Example: Setting basic and detailed trace levels	275
Example: Setting specific trace points	276
Example: Excluding specific trace points	276
Example: Resetting to the initial trace settings	276
Example: Turning off tracing	277
Example: Sending the trace to sysprint	277
Example: Displaying servants	277
Example: Displaying trace settings and Java string trace settings	277
Example: Displaying JVM heap information	278
Example: Displaying sessions	279
Example: Displaying status of a server	280
Example: Displaying status of clusters	280
Example: Getting help for the modify command	281
Example: Modifying the Java trace string	282
Display command	282
Example: Displaying active replies	282
Example: Displaying active address spaces	282
Example: Displaying the status of address spaces registered for automatic restart management	283
Example: Displaying units of work (transactions) for the Information Management System	283

Chapter 11. Welcome to Programming 285

Chapter 12. Deploying and managing using programming. 287

Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs	287
Developing an administrative client program	288
Extending the WebSphere Application Server administrative system with custom MBeans	293
Java 2 security permissions	295
Java Management Extensions MBean multiprocess model request flow for WebSphere Application Server for z/OS	295

Chapter 13. Welcome to Data 307

Chapter 14. Working with server configuration files 309

Configuration documents	309
Configuration document descriptions	311
Object names	313
Configuration repositories	313
Handling temporary configuration files resulting from session timeout	314
Changing the location of temporary configuration files	314
Changing the location of backed-up configuration files	315
Changing the location of temporary workspace files	315
Backing up and restoring administrative configurations	316
Server configuration files: Resources for learning	316
Backing up the WebSphere for z/OS system	316

Chapter 15. Welcome to Administrative agents. 319

Chapter 16. Managing administrative agents 321

Cells	321
Configuring cells	321
Cell settings	322
Deployment managers	323
Configuring deployment managers	323
Deployment manager settings	323
Node	324
Managing nodes	324
Node collection.	326
Administration service settings	327
Standalone	327
Preferred Connector	327
Extension MBean Providers collection	327
Repository service settings	328
Audit Enabled	328
Node agents.	329
Managing node agents	329
Node agent collection	329
Remote file services	331
Configuring remote file services	332

File transfer service settings	332
File synchronization service settings	333
Administrative agents: Resources for learning	334
Location service daemons	335
Determining if the location service daemon is running.	336
Steps for stopping or canceling the location service daemon from the MVS console	336

Modifying location service daemon settings	336
z/OS location service daemon settings	337

Notices 339

Trademarks and service marks 341

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.

Chapter 1. Welcome to System Administration

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 Chapter 2, “Welcome to the Administrative Console,” on page 3.

- **Scripting**

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

For more information, refer to Chapter 6, “Welcome to Scripting,” on page 25.

- **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 Chapter 8, “Welcome to Commands,” on page 245.

- **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 Chapter 11, “Welcome to Programming,” on page 285.

Data

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

For more information, refer to Chapter 13, “Welcome to Data,” on page 307.

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 Chapter 15, “Welcome to Administrative agents,” on page 319.

Chapter 2. Welcome to the Administrative Console

Use both the MVS console and the Application Server administrative console to administer the Application Server. For example:

- Use MVS commands issued from the MVS console to start the base application server controller region, and the node agent and deployment manager.
- In an application server configuration, you must start the first server with an MVS operator command. Once the first server is started, you can then use the administrative console, if it has this application, to start other application servers in the node. Once the Deployment Manager and Node Agent are active (in an ND configuration), you can use the administrative console to start and stop application servers.
- WLM starts all servant regions using Address Space Create (ASCRE) The administrative console allows you to display and modify Application Server applications and the environment in which they run.

Chapter 3. Identifying where to perform WebSphere Application Server operations

Administering WebSphere Application Server involves the use of both the MVS console and the WebSphere Application Server administrative console. For example:

- Use MVS commands issued from the MVS console to start the base Application Server control region, the network deployment node agent, and the deployment manager.
- In a base Application Server configuration, you must start the first server with an MVS operator command. Once the first server starts, you can then use the administrative console, if it has this application, to start other Application Servers in the node. Once the deployment manager and node agent are active (in a network deployment configuration), you can use the administrative console to start and stop application servers.
- Workload management starts all servant regions using Address Space Create (ASCRE).

The following table lists the main Application Server operations tasks and directs you to information that helps you to perform these tasks. The Application Server activities and operations can be performed from:

- A z/OS or OS/390 MVS console (most operations)
- The Application Server administrative console (some operations)
- TSO or resource recovery services (RRS) panels (some operations).

(Some link titles are split for publication.)

Table 1. Application Server operations tasks

Task	MVS console	Application Server administrative console	TSO panel	Reference to associated procedure
Start operations				
Starting the Application Server environment and location service daemon	Yes	No	No	See Starting servers.
Starting a cluster or application server	Yes	Application server only	No	See Starting clusters, Starting servers, and Using the administrative console.
Stop operations				
Stopping the location service daemon	Yes	No	No	See Steps for stopping or canceling the location service daemon from the MVS console .
Stopping a cluster	Yes	Yes	No	See Stopping clusters.

Table 1. Application Server operations tasks (continued)

Task	MVS console	Application Server administrative console	TSO panel	Reference to associated procedure
Stopping an application server	Yes	Yes	No	See Stopping servers.
Cancel operations				
Canceling the location service daemon	Yes	No	No	See Steps for stopping or canceling the location service daemon from the MVS console.
Canceling a cluster	Yes	Yes	No	See Stopping clusters.
Canceling an application server	Yes	Yes	No	See Stopping servers.
Display operations				
Displaying the status of ARM-registered address spaces including clusters and servants	Yes	No	No	See Displaying the status of ARM-registered address spaces including WebSphere for z/OS servers and server instances.
Displaying units of work (threads) for DB2	Yes	No	No	See the <i>DB2 Universal Database for OS/390 and z/OS Command Reference</i> manual.
Displaying indoubt units of work (threads) for DB2	Yes	No	No	See the <i>DB2 Universal Database for OS/390 and z/OS Command Reference</i> manual.
Displaying units of work for RRS	No	No	Yes	See the <i>z/OS MVS Programming: Resource Recovery</i> manual.
Displaying units of work for CICS	Yes	No	Yes	See the <i>CICS Operations and Utilities Guide</i> manual.
Displaying units of work (transactions) for IMS	Yes	No	No	See the <i>IMS/ESA Summary of Operator Commands</i> manual.
Displaying the status of a cluster	Yes	Yes	No	See Example: Displaying status of clusters .
Displaying the status of a server	Yes	Yes	No	See Example: Displaying status of a server .

Table 1. Application Server operations tasks (continued)

Task	MVS console	Application Server administrative console	TSO panel	Reference to associated procedure
Displaying active address spaces	Yes	No	No	See Example: Displaying active address spaces.
Displaying active replies	Yes	No	No	See Example: Displaying active replies.
Modify operations				
Getting help for the modify command	Yes	No	No	See Example: Getting help for the modify command.
Canceling application clusters and servers	Yes	No	No	See Example: Canceling application clusters and servers with the modify command.
Modifying trace variables dynamically	Yes	No	No	See Example: Setting basic and detailed trace levels.
Modifying the Java trace string	Yes	No	No	See Example: Modifying the Java trace string.
Displaying status	Yes	No	No	See Modify command.
Other Application Server operations				
ARM and restart	Yes	No	No	See Automatic restart management.
Setting up error log streams for different clusters and servants	No	You can associate a log stream with a cluster from the administrative console	No	See Setting up the error log.
Setting up System Management Facilities recording	Yes	Enable it from here, but initiate it from the MVS console.	No	See Collecting job-related information with Systems Management Facility (SMF).
Shutting down the WebSphere for z/OS environment	Yes	Application server only	No	See Stopping clusters, Stopping servers, and Steps for stopping or canceling the location service daemon from the MVS console.

Table 1. Application Server operations tasks (continued)

Task	MVS console	Application Server administrative console	TSO panel	Reference to associated procedure
Taking a WebSphere for z/OS system cluster out of service	Yes	Application server only; You cannot take a WebSphere for z/OS system cluster out of service from the administrative console	No	
Workload Management				
Displaying the status of a WLM application environment	Yes	No	No	See Handling workload management and server failures.
Handling workload management and server failures	Yes	No	No	See Handling workload management and server failures.
Getting out of the stopped state and back to the available state	Yes	No	No	See Handling workload management and server failures.
Checking and managing the workload management application environment (display, stop/quiesce, restart/resume)	Yes	No	No	See Handling workload management and server failures and WLM dynamic application environment operator commands.

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. 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. After you point a Web browser at the URL for the administrative console, enter your user ID and, if needed, a password on a Login page.
4. Browse the administrative console.
5. Specify console preferences.
6. Access help.

Starting and stopping the administrative console

The administrative console application is installed during the initial installation process. 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. Verify that the application server for the administrative console is running. Verify that the administrative console runs on the deployment manager (dmgr) application server for the WebSphere Network Deployment product. Issue the **startManager** command at the MVS console to start the deployment manager.
 - b. Enable cookies in the Web browser that you use to access the administrative console in order for the administrative console to work correctly.
 - c. In the same Web browser, type
`http://your_fully_qualified_server_name:9090/admin` where *your_fully_qualified_server_name* is the fully qualified host name for the machine that contains the administrative server. If security is enabled, your request is redirected to
`https://your_fully_qualified_server_name:9043/admin` where *your_fully_qualified_server_name* is the full 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>

Note: The URL appears on two lines for printing purposes. Enter the URL on one line in your browser.

- d. Wait for the console to load into the browser.

If you cannot start the administrative console because the console port conflicts with an application already running on the machine, change the port number in the

```
install_root/config/cells/cell_name/  
nodes/node_name/servers/server_name/server.xml
```

and `install_root/config/cells/cell_name/virtualhosts.xml` files. Change all occurrences of port 9090 (or whatever port was selected during installation of WebSphere Application Server) to the desired port for the console. Alternatively, shut down the other application that uses the conflicting port before starting the WebSphere Application Server product.

Note: The directory path for the `server.xml` file appears on two lines for printing purposes.

2. A Login page appears after the console starts. Log into the console.
 - a. Enter your user name (or user ID).

Changes made to server configurations are saved to the user ID. Server configurations also are saved to the user ID if there is a session timeout. A user ID lasts for the duration of the session for which it was used to log in. If you enter an ID that is already in use (and in session), you are prompted to do one of the following:

 - Force the existing user ID out of session. The configuration file used by the existing user ID is saved in the temp 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 done the following things for the console:
 - 1) Specified security user IDs and passwords
 - 2) Enabled global security
 - c. Click **OK**.
3. Stop the administrative console. Click **Save** on the console taskbar to save work that you have done and then click **Logout** to exit the console.

Note that 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 also specify a password.

Specifying a user allows you to resume work done previously with the product. After you type in a user ID, click **OK** to proceed to the next page and access the administrative console.

To view this page, start the administrative console.

User ID

Specifies a string identifying the user. The user ID must be unique to the administrative server.

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

If you are logging in from a client machine that currently has a browser open on the administrative console and you connect to the same server, your two connections might share the same HTTP session object, whether or not you are

logging in under different user IDs. You must use two different types of browsers (for example, Microsoft Internet Explorer and Netscape Communicator) to log in to the administrative console from the same client machine in order to work from two different HTTP session objects, whether or not you use the same user ID. That is, if you need two concurrent sessions on the same client machine, access the administrative console from two different browser types.

Note that you can use the same browser type to log in twice to the console, but you must log in from different physical hosts and use different user IDs. For example, suppose that Joe logs into the administrative console under the user ID *joe* on the physical host *myhost1* using an Internet Explorer browser. Joe can log into the same console a second time using an Internet Explorer browser, but he must use a different user ID (such as *peggy*) and a different physical host (such as *myhost2*).

Data type String

Another user is currently logged in with the same user name

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

This field appears 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 accessing the administrative console.

Work with the master configuration

When enabled, specifies that you want to use the default administrative configuration and not use the configuration last used for the user's session. Changes made to the user's session since the last saving of the administrative configuration will be lost.

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

Data type Boolean
Default false

Recover changes made in prior session

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

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

Data type Boolean
Default true

Administrative console areas

Use the administrative console to create and manage resources, applications and servers or to view product messages.

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

The console has the following main areas, which you can resize as desired:

Taskbar

The taskbar offers options for returning to the console Home page, saving changes to administrative configurations, specifying console-wide preferences, logging out of the console, and accessing product information.

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 on a + beside a tree folder or item to expand the tree for the folder or item. Click on a - to collapse the tree for the folder or item. Click on 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 on configured objects to view their configurations, run-time status, and options. Click **Home** in the taskbar to display the workspace Home page, which contains links to information on using the WebSphere Application Server product.

WebSphere Status

The status messages area at the bottom of the console provides information on messages returned by the WebSphere Application Server. The information deals with problems in your administrative configuration as well as messages about run-time events. Click **Previous** or **Next** to toggle among displays of configuration problems and run-time events.

You can adjust the interval between automatic refreshes in the Preferences settings.

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 have enabled.

- **Abort.** Aborts a transaction that is not yet in the prepared state. All operations that the transaction completed are undone.
- **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 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. Note that the administrative console does not support using the Back and Forward buttons of a browser. Using those buttons can cause intermittent problems. Use the Back or Cancel buttons on the administrative console panels instead.

- **Browse.** Opens a dialog that enables you to look for a file on your system.
- **Cancel.** Exits the current page or dialog, discarding unsaved changes. Note that the administrative console does not support using the Back and Forward buttons of a browser. Using those buttons can cause intermittent problems. Use the Cancel button on the administrative console panels instead.
- **Change.** In the context of security, lets you search the user registry for a user ID for an application to run under. In the context of container properties, lets you change the data source 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 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 tabbed pages in a dialog and exits the dialog.
- **Delete.** Removes the selected instance.
- **Details.** Shows the details about a transaction.
- **Done.** Saves your changes to all tabbed pages in a dialog and exits the dialog.
- **Down.** Moves downward through a list.
- **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.
- **Export.** Accesses a page for exporting EAR files for an enterprise application.
- **Export DDL.** Accesses a page for exporting DDL files for an enterprise application.
- **Export Keys.** Exports LTPA keys to other domains.
- **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 this button 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 has been made to the cell repository that needs to be replicated to that node. Clicking this button clears all synchronization optimization settings and performs configuration synchronization anew, so there will be no mismatch between node and cell configuration after this operation is performed. This operation can take a while to perform.
- **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. If new keys need to be generated, use this button after the server is back up with security turned on. Clicking this button 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 would allow in-flight requests to complete before shutting down the whole server process. This shutdown mode is faster than the normal server stop processing, but some application clients may receive exceptions.
- **Import Keys.** Imports new LTPA keys from other domains. To support single sign on (SSO) in WebSphere across multiple WebSphere domains (cells), LTPA keys and a password should be shared among the domains. After exporting the keys from one of the cells into a file, clicking this button imports the keys into all 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 install page, which you use to deploy an application or EJB or Web component onto an application server.
- **Install RAR.** The Install RAR button opens a dialog used to install a JCA connector and create a resource adapter for it.
- **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 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.
- **New.** Displays a page which you use to define a new instance. For example, clicking **New** on the Application Servers page displays a page on which 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, discarding any unsaved changes.
- **Refresh.** Refreshes the view of data for instances currently listed on this tabbed page.
- **RegenerateKey.** 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 Node.** Deletes the selected node.
- **Reset.** Clears your changes on the tab or page and restores the most recently saved values.
- **Restart all Servers on Node.** Stops all application servers on the node and starts them again.
- **Retrieve new.** Retrieves a new record.
- **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 now in use.
- **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 this button 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 has been made to the cell repository that needs to be replicated to that node. Clicking this button requests that a node synchronization operation be 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. So 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 that cannot be stopped by the Stop or Immediate Stop commands. Some application clients may receive exceptions. Always attempt Immediate Stop before using this option.
- **Test Connection** After you have defined and saved a data source, you can click this button 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 all at once.
- **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 deployed on a server with an updated application. As part of the updating, you might need to complete steps on the Preparing for application install 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 OMG Common Secure Interoperability (CSI) protocol.
- **Use Cell SAS.** Enables IBM Secure Authentication Service (SAS).
- **Use Cell Security.** Enables cell security.
- **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, Filter, and Preferences

These are described in “Administrative console scope settings” on page 20, “Administrative console filter settings” on page 20, and “Administrative console preference settings” on page 19.

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 in “Administrative console buttons” on page 12. 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.

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.

Buttons for performing actions

The available buttons are described in “Administrative console buttons” on page 12.

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 tree are shown below. To use the tree, expand a main topic of interest and select an item from the expanded list to display a page on which you can perform the administrative task.

Servers

Enables you to configure application servers and other types of servers such as JMS servers.

Applications

Enables you to install applications onto servers and manage the installed applications.

Resources

Enables you to configure resources and to view information on resources existing in the administrative cell.

Security

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

Environment

Enables you to configure hosts, Web servers, WebSphere Application Server variables and other components.

System Administration

Enables you to manage components and users of a Network Deployment product.

Troubleshooting

Enables you to check for configuration errors and problems, view log files, and enable and disable tracing.

Administrative console taskbar actions

Use the taskbar of the administrative console to return to the console Home page, save changes that you have made to administrative configurations, specify console preferences, log out of the console, and access product information.

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 several actions.

Home

Displays the administrative console Home page in the workspace. The workspace is on the right side of the console. The Home page provides links to information on using the WebSphere Application Server product.

Save

Displays the Save page, which you use to save work that you have done in the administrative console.

Changes made to administrative configurations are saved to a master repository comprised of XML configuration files. To see what files are affected by the changes, expand **View items with changes**. Periodically save your work to ensure that you do not inadvertently lose changes that you have made.

Preferences

Displays the Preferences page, in which you can specify whether you want the administrative console workspace page to refresh automatically after changes, confirmation dialogs to display, and the default scope to be the administrative console node.

Logout

Logs you out of the administrative console session and displays the Login page. If you have changed the administrative configuration since last saving the configuration to the master repository, the Save page displays before returning to the Login page. Click **Save** to save the changes to the master repository, **Discard** to exit the session without saving changes, or **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 on online help for the WebSphere Application Server product.

Hide/Show Field and Page Descriptions toggle

Enables you to select whether information on console pages and fields within the pages is shown. Icons on the right-hand side of the taskbar provide the toggle.

WebSphere status settings

Use the WebSphere Status area of the administrative console to view error and run-time event messages returned by WebSphere Application Server.

The WebSphere Status area displays along the bottom of the console and remains visible as you navigate from the WebSphere Home page to other pages. The area displays two frames: **WebSphere Configuration Problems** and **WebSphere Runtime Messages**. Click **Previous** or **Next** to toggle between the frames.

Click the icon in the upper-right of the area to refresh the information displayed. You can adjust the interval between automatic refreshes in the **Preferences** settings.

WebSphere Configuration Problems

Displays the number of workspace files. This frame also displays the number of problems with the administrative configuration for the user ID.

Click on the number to view detailed information on the problems.

WebSphere Runtime Messages

Displays the number of messages returned by WebSphere Application Server as well as the number of error messages(x icon), warning messages (! icon), and informational messages (i icon).

Click on the number of messages to view details.

Specifying console preferences

Throughout the administrative console are pages that have **Preferences**, **Filter**, and **Scope** fields near the top of the pages. To customize how much data is shown, select options in **Preferences**, **Filter** and **Scope**.

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

1. Go to the navigation tree of the administrative console and select **Applications > Enterprise Applications**.
2. Expand **Preferences**.
3. For the **Maximum rows** field, specify the maximum number of rows to be displayed when the collection is large. The default is 25. Rows that exceed the maximum number will appear on subsequent pages.
4. For the **Filter history** field, place a checkmark beside **Retain filter criteria** if you want the last filter criteria entered in the filter function to be retained. When you return to the Applications page, the page will initially use the retained filter criteria to display the collection of applications in the table below. Otherwise, remove the checkmark beside **Retain filter criteria** to not retain the last filter criteria.
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 checkmark beside) **Retain filter criteria**.

Other pages have similar fields in which you specify console preferences. While **Preferences**, **Filter** and **Scope** control how much data is shown in the console, the **Preferences** taskbar option controls general behavior of the console. Click **Preferences** on the console taskbar to view the Preferences page. Additionally, you can select whether information on console pages and fields within console pages is shown using the **Hide Field and Page Descriptions toggle**. Icons on the right-hand side of the taskbar provide the toggle.

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 **Preferences** on the console taskbar.

Enable 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's name in the collection of servers.

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

Data type	Boolean
Default	true

Do not confirm WorkSpace Discards

Specifies whether the confirmation dialog is displayed after a request to discard the workspace. The default is to display confirmation dialogs.

Data type	Boolean
Default	false

Use Default Scope

Specifies whether the default scope is the administrative console node. The default is for the scope not to be the console node.

Data type	Boolean
Default	false

Hide/Show Banner

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

Data type	Boolean
Default	true

Hide/Show Descriptions

Specifies whether information on console pages and fields within the pages is shown. The Hide/Show Field and Page Descriptions icons on the right of the taskbar provide the same function. The default is to show page and field descriptions.

Data type	Boolean
Default	true

Administrative console preference settings

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

Maximum rows

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

Filter history

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 later, retained filter criteria control the application collection that appears in the table.

Administrative console filter settings

Use the filter settings to specify how to filter entries shown in a collection view.

Select the column to filter, then enter the filter criteria.

Column to filter

Use the drop-down list to select the column to filter. When you apply the filter, the collection table is modified accordingly.

For example, select **Application Servers** if you plan to enter criteria by which to filter application server names.

Filter criteria

In the field beside the drop-down list, enter a string that must be found in the name of a collection entry to qualify the entry to appear in the collection table. The string can contain %, *, or ? symbols as wildcard characters. For example, enter *App* to find any application server whose name contains the string *App*.

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

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

Administrative console scope settings

Use Scope settings to filter the contents of an administrative console collection table to a particular cell, node, or server. By changing the value for **Scope** you can see other variables that apply to an object 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.

Cell Limits the visibility to all servers on the named cell.

Node Limits the visibility to all servers on the named node.

Server Limits the visibility to the named server.

The server scope has precedence over the node and cell scopes, and the node scope has precedence over the cell scope. Note that objects are created at only one scope, though they might be visible at more than one scope.

Accessing help

The WebSphere Application Server product provides information on using the product and information about specific pages and fields.

1. To view information on console pages and fields within console pages, enable the **Hide Field and Page Descriptions toggle**. Icons on the right-hand side of

the console taskbar provide the toggle. Click on the **i** icon beside a field or page description to view help (reference) information.

2. Access product information.

- Click **Home** on the administrative console taskbar and select a link to information on using the WebSphere Application Server product. The link to the information center provides concept, task and reference information.
- Click **Help** on the administrative console taskbar and select from the list of reference topics. This same information can be accessed by clicking on the **i** icon beside a field or page description.

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

- Listing of all IBM WebSphere Application Server Redbooks
- System Administration for WebSphere Application Server V5 -- Part 1: Overview of V5 Administration

Chapter 5. Using the MVS console

Use the MVS console to manage the IBM WebSphere Application Server product as well as the Network Deployment product.

1. See the *z/OS MVS System commands* manual at <http://www.ehonet.ibm.com/public/applications/publications/cgi-bin/pbi.cgi> for information on how to use MVS operator commands.
2. Optionally use standard console automation products to automate WebSphere for z/OS operations. All automations for the WebSphere for z/OS environment are done using interfaces from the MVS console. Products such as Netview are presented copies of messages that are to be displayed on the MVS console. These automation products can also enter commands into the system using a "virtual" MVS console as a source.

Chapter 6. Welcome to Scripting

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

See the **Reference > Scripting interfaces** section of the information center navigation. It provides many examples of how to perform tasks using the scripting interface.

The scripting program supports the same tasks as the administrative console, such as the ability to:

- 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 J2EE resource providers for applications requiring 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

Chapter 7. Deploying and managing using scripting

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

To configure and manage WebSphere Application Server, you can also use the 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*. There are also several command-line tools that you can use to start, stop, and monitor WebSphere Application Server processes and nodes. These tools only work on local servers and nodes. They cannot operate on a remote server or node. See *Managing using command-line tools* for more information.

1. Launch a scripting client interactively, as an individual command, in a script, or in a profile.
2. Perform one of the following options:
 - Install applications into runtime.
 - Edit applications.
 - Display installed applications.
 - Uninstall applications.
 - Identify attributes and operations for running objects.
 - Modify and query running object attributes.
 - Invoke operations on running objects.
 - Modify the configurations.
 - Modify nested attributes.

Migrating from Systems Management Scripting API (SMAPI) V4 to wsadmin V5

The purpose of this section is to provide guidance for migrating from WebSphere Application Server V4 Systems Management Scripting Application Programming Interface (SMAPI) to wsadmin in V5.

The SMAPI feature of WebSphere Application Server V4 provides the same functionality as the graphical user interface to Systems Management (systems management enhanced user interface). The only scripting language supported by the SMAPI is REXX. The SMAPI provides predefined REXX functions that allow the user to perform administration functions (CB390FCG) and operation functions (CB390CMD). There are also functions to generate input (XMLGEN) and parse the output (XMLPARSE, XMLFIND, and XMLEXTRACT) from the administration and operations applications.

The wsadmin client scripting tool in WebSphere Application Server V5 replaces the SMAPI of WebSphere Application Server V4. Version 5 distinguishes between functions that modify or query the Systems Management configuration from functions that control runtime operational commands. The wsadmin scripting client uses the Bean Scripting Framework (BSF) and is based on Java™

Management Extensions (JMX). Java™ Command Language (JACL) is the supported scripting language. The wsadmin tool provides the ability to run scripting commands interactively from a wsadmin shell, from the command line, or from a script. The wsadmin tool operates on configurations and running objects through the following set of management objects: AdminConfig for configuration commands, AdminControl for operational commands, AdminApp for application management commands, and Help. Each object has commands that you use to perform administrative tasks. The wsadmin tool requires that you specify a scripting object, a command, and command arguments.

This article shows how to create WebSphere Application Server V5 scripts that perform actions similar to those performed by V4 SMAPI.

1. Identify the SMAPI command type in the script, configuration or operation.
 - Configuration command types use the CB390CFG command. You can list, create, delete, and change various configuration objects like the conversation, system, servers, Java™ platform Enterprise Edition (J2EE) resources, and so on with this command.
 - Operation command types use the CB390CMD command. You can start, stop, cancel, restart. You can also list defined server instances.
2. Find the corresponding V5 objects that the V4 configuration command can be performed on. In V5, the AdminConfig management object uses the create, list, modify, remove, show, and showAttribute commands on configuration objects. V5 configuration objects include Server, ModuleDeployment, Node, ServerCluster, URL, VirtualHost, and URLProvider in addition to the ones listed in the following table.

V4 SMAPI configuration object	V5 wsadmin configuration type
Conversation	Not applicable
System	Not applicable
Sysplex	Not applicable
Server	Not applicable
J2EE server	ServerGroup
J2EE Application	use AdminApp object
Server Instance	Server
Container	Not applicable
LRM	Not applicable
LRMI	Not applicable
Container/LRM associations	Not applicable
Application Family	Not applicable
J2EE resources/J2EE resource Instances	J2CConnectionFactory/J2CResourceAdapter /JMSCConnectionFactory /JMSDestination/JMSProvider /MailSession/WAS40DataSource

3. Determine the V5 attribute names by using the online help commands of the AdminConfig object. Examples include attributes, defaults, parents, required, or types.
4. Convert application installation commands. For application installation, use the AdminApp object installInteractive command to complete a successful

installation. Then locate message WASX7278I in the wsadmin.traceout log file. Use the data in the message to construct an installation command for your source.

- Convert operational commands. The following illustrates an example:

V4 SMAPI action	V5 wsadmin object and command
CB390CMD("-action 'start' -servername 'BBOASR1' -serverinstancename '*' -output 'FCT33'")	\$AdminControl startServer server1

- Save configuration changes. In V5, configuration changes are made to a temporary workspace. These changes are not committed to the WebSphere Application Server configuration until you invoke the save command on the AdminConfig object. If your script makes configuration changes, for example, creates, removes, or changes objects, or installs or uninstalls applications, invoke the following command to commit the change:

```
$AdminConfig save
```

Example: Migrating - Starting an application server

The following examples demonstrate how to start an application server with WebSphere Application Server V4 and V5:

- V4 SMAPI REXX script

```
call syscalls 'ON'
signal on error

rc = 0
rc = CB390CMD("-action 'start' -servername 'BBOASR1'
-serverinstancename '*' -output 'FCT33'")
if (rc == 4) then
do say "FCT Test #33 failed"
Exit
end

rc = XMLPARSE("FCT33" "ALL")
if (rc == 4) then
do say "FCT Test #33 failed while XMLPARSE"
Exit
end
say "FCT Test #33 completed"
exit

error:
say "Error in FCT Test #33" rc "at line" sigl
say sourceline(sigl)
exit
```

- V5 wsadmin tool

The command sequence given below accomplishes approximately the same thing as the V4 SMAPI commands.

If you are connected to a server in a base installation, you cannot request to start another one. You can only start an application server if you have a Network Deployment installation. In a Network Deployment installation, use the following:

```
wsadmin> $AdminControl startServer server1 mynode
```

When the wsadmin scripting tool successfully starts a WebSphere Application Server, you see a message similar to the following:

```
WASX7262I: Start completed for server "server1" on node "mynode"
```

Related reference

Example: Migrating - Installing an application

The following examples demonstrate how to install an application in the WebSphere Application Server V4 and V5:

- V4 SMAPI REXX script

```
call syscalls 'ON'
signal on error
say "FCT Test #processearfile"

name. = 0
name.1 = "conversationname"
name.2 = "j2eeservername"
name.3 = "earfilename"
val. = 0
val.1 = "API Functiontest"
val.2 = "BBOASR4"
val.3 = "/tmp/filetransfer_resolved.ear"

rc = 4
i = 1

/* Generate XML Input */
do while(name.i <> '0')
  rc = XMLGEN("tempin" name.i val.i)
  if (rc == 4) then
    do say "FCT Test #processearfile failed while XMLGEN"
    exit
    end
  i = i+1
end;

/* Call the function: processearfile */
rc = CB390CFG("-action 'processearfile'
-xmlinput 'inputprocessearfile.xml' -input 'tempin'
-output 'processearfile'")
if (rc == 4) then
  do say "FCT Test #processearfile failed"
  exit
  end

/* Parse the result */
rc = XMLPARSE("processearfile" "ALL")
if (rc == 4) then
  do
    say "FCT Test #processearfile failed while XMLPARSE"
  exit
  end

say "FCT Test #processearfile completed"
return 0
exit

error:
say "Error in FCT Test #processearfile" rc "at line" sigl
say sourceline(sigl)
exit
```

- V5 wsadmin tool

The command sequence given below accomplishes approximately the same thing as the V4 SMAPI commands.

- Start the wsadmin scripting tool:

```
$WAS_ROOT/AppServer/bin:> ./wsadmin.sh
```

When the wsadmin scripting tool successfully starts, you see a message similar to the following:

```
WASX7209I: Connected to process "server1" on node
SY1 using SOAP connector; The
  type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"
```

- Construct the attribute string:

```
wsadmin> set attrs [list -node SY1 -server server1]
```

- Install the application:

```
wsadmin> $AdminApp install ../installableApps/filetransfer.ear $attrs
```

When the application successfully installs, you see a series of messages similar to the following:

```
WASX7327I: Contents of was.policy file:
  grant codeBase "file:${application}" {
    permission java.security.AllPermission;
  };
```

```
ADMA5016I: Installation of WebSphere File Transfer
Servlet Application started.
```

```
ADMA5005I: Application WebSphere File Transfer Servlet
Application configured in
WebSphere repository
```

```
ADMA5001I: Application binaries saved in /WebSphere/V5R0M0/
AppServer/wstemp/Scriptf3964b86cb/workspace/cells/SY1/
applications/WebSphere File Transfer Servlet Application.ear/
WebSphere File Transfer Servlet Application.ear
```

```
ADMA5011I: Cleanup of temp dir for app WebSphere File
Transfer Servlet Application done.
```

```
ADMA5013I: Application WebSphere File Transfer Servlet
Application installed successfully.
```

- Save your changes:

```
wsadmin> $AdminConfig save
```

You can use the AdminApp taskInfo command to obtain information about each task option:

```
$AdminApp taskInfo ../installableApps/filetransfer.ear MapWebModToVH
```

The taskInfo command results in information similar to the following:

```
MapWebModToVH: Selecting Virtual Hosts for Web Modules
Specify the virtual host where you want to install the
Web modules contained in
your application. Web modules can be installed on the same
virtual host or dispersed among several hosts.
```

```
WASX7348I: Each element of the MapWebModToVH task consists of
the following 3 fields: "webModule", "uri", "virtualHost".
Of these fields, the following may be assigned values: "virtualHost"
and the following are required: "virtualHost"
```

The current contents of the task after running default bindings are:

```
webModule: WebSphere Admin File Transfer Application
uri: filetransfer.war,WEB-INF/web.xml
virtualHost: admin_host
```

You can use the AdminApp installInteractive command to step through all the installation tasks, one at a time. If you use the installInteractive command to successfully install an application, an option string logs in the wsadmin.traceout file under the message ID WASX7278I. You can copy and paste this option string into wsadmin scripts.

Related reference

“Example: Listing running applications on running servers using wsadmin” on page 210

“Example: Starting an application using wsadmin” on page 212

“Example: Stopping running applications on a server using wsadmin” on page 212

Example: Migrating - Uninstalling an application

The following examples demonstrate how to uninstall an application in the WebSphere Application Server V4 and V5:

- V4 SMAPI REXX script

```
call syscalls 'ON'
signal on error
say "FCT Test #processearfile"

name. = 0
name.1 = "conversationname"
name.2 = "j2eeservername"
name.3 = "earfilename"
val. = 0
val.1 = "API Functiontest"
val.2 = "BBOASR4"
val.3 = "/tmp/filetransfer_resolved.ear"

rc = 4
i = 1

/* Generate XML Input */
do while(name.i <> '0')
  rc = XMLGEN("tempin" name.i val.i)
  if (rc == 4) then
    do say "FCT Test #processearfile failed while XMLGEN"
    exit
  end
  i = i+1
end;

/* Call the function: processearfile */
rc = CB390CFG("-action 'processearfile'
-xmlinput 'inputprocessearfile.xml' -input 'tempin'
-output 'processearfile'")
if (rc == 4) then
  do say "FCT Test #processearfile failed"
  exit
end

/* Parse the result */
rc = XMLPARSE("processearfile" "ALL")
if (rc == 4) then
  do
    say "FCT Test #processearfile failed while XMLPARSE"
  exit
end

say "FCT Test #processearfile completed"
return 0
exit

error:
say "Error in FCT Test #processearfile" rc "at line" sigl
say sourceline(sigl)
exit
```

- V5 wsadmin tool

The command sequence given below accomplishes approximately the same thing as the V4 SMAPI commands.

- List the installed applications:

```
wsadmin> $AdminApp list
```

When the wsadmin scripting tool lists the installed applications, you see output similar to the following:

```
WebContainerTestApplication
  adminconsole
  Greeting
  IvtApp
```

- Uninstall the IvtApp application:

```
wsadmin> $AdminApp uninstall ivtApp
```

When the application successfully uninstalls, you see a series of messages similar to the following:

```
ADMA5017I: Uninstallation of ivtApp started.
ADMA5104I: Server index entry for SY1 was updated successfully.
ADMA5102I: Deletion of config data for ivtApp
from config repository completed successfully.
ADMA5011I: Cleanup of temp dir for app ivtApp done.
ADMA5106I: Application ivtApp uninstalled successfully.
```

- Save your changes:

```
wsadmin> $AdminConfig save
```

Related reference

“Example: Listing running applications on running servers using wsadmin” on page 210

“Example: Starting an application using wsadmin” on page 212

“Example: Stopping running applications on a server using wsadmin” on page 212

Example: Migrating - Defining a Java™ 2 Enterprise Edition resource

The following examples demonstrate how to define a Java™ 2 Enterprise Edition (J2EE) resource in the WebSphere Application Server V4 and V5:

- V4 SMAPI REXX script

```
call syscalls 'ON'
signal on error

name. = 0
name.1 = "conversationname"
name.2 = "j2eeresourcenam"
name.3 = "j2eeresourcedescription"
name.4 = "j2eeresourcetype"
val. = 0
val.1 = "API Functiontest"
val.2 = "db2resA"
val.3 = "my DB2 resource"
val.4 = "DB2datasource"

rc = 4
i = 1
do while (name.i <> '0')
  rc = XMLGEN("tempin" name.i val.i)
  if (rc == 4) then
    do say "create J2EE resource failed during XMLGEN"
  exit
end
```

```

i = i+1
end;

rc = CB390CFG("-action 'createj2eeresource'
-xmlinput 'inputcreatej2eeresource.xml' -input 'tempin'
-output 'test.create.resource'")
if (rc == 4) then
  do say "create J2EE resource failed"
  exit
end
exit
error:
say "script error at line" sigl
say sourceline(sigl)
exit

```

- V5 wsadmin tool

The command sequence given below accomplishes approximately the same thing as the V4 SMAPI commands.

- Set the node:

```
wsadmin> set node [$AdminConfig getid /Node:SY1/]
```

When the wsadmin scripting tool sets the node, you see output similar to the following:

```
SY1(cells/SY1/nodes/SY1:node.xml#Node_1)
```

- Set the provider:

```
wsadmin> set provider [$AdminConfig create JDBCProvider
$node {{classpath /db2beta/db2710/class
es/db2j2classes.zip}
{implementationClassName com.ibm.db2.jcc.DB2ConnectionPoolDataSource}
{name myDB2driver}}]
```

When the application successfully sets the provider, you see a message similar to the following:

```
myDB2driver(cells/SY1/nodes/SY1:resources.xml#JDBCProvider_3)
```

- Create the J2EE resource:

```
wsadmin> $AdminConfig create DataSource $provider {{name db2resA}
{description My DB2 resource}}
```

When the wsadmin scripting tool creates the J2EE resource, you see output similar to the following:

```
db2resA(cells/SY1/nodes/SY1:resources.xml#DataSource_2)
```

- Save your changes:

```
wsadmin> $AdminConfig save
```

Example: Migrating - Stopping an application server

The following examples demonstrate how to stop an application server with WebSphere Application Server V4 and V5:

- V4 SMAPI REXX script

```

call syscalls 'ON'
signal on error

rc = 0
rc = CB390CMD("-action 'stop' -servername 'BBOASR4'
-serverinstancename '*' -output 'FCT34'")
if (rc == 4) then
  do say "FCT Test #34 failed"
  exit
end
rc = XMLPARSE("FCT34" "ALL")

```

```

if (rc == 4) then
  do say "FCT Test #34 failed while XMLPARSE"
  exit
end
say "FCT Test #34 completed"
exit

```

```

error:
say "Error in FCT Test #34" rc "at line" sigl
say sourceline(sigl)
exit

```

- V5 wsadmin tool

The command sequence given below accomplishes approximately the same thing as the V4 SMAPI commands.

Stop the application server:

```
wsadmin> $AdminControl stopServer server1 mynode
```

When the wsadmin scripting tool successfully stops a WebSphere Application Server, you see messages similar to the following:

```

WASX7337I: Invoked stop for server "server1"
Waiting for stop completion.
  WASX7264I: Stop completed for server
"server1" on node "mynode"

```

Related reference

“AdminControl object for scripted administration” on page 75

Launching scripting clients

You can run scripting commands in several different ways. The command for invoking a scripting process is located in the `/WebSphere/V5R0M0/AppServer/bin` directory or the `/WebSphere/V5R0M0/DeploymentManager/birectory`. To invoke a scripting process, use the `wsadmin.sh` file.

To specify the method for executing scripts, perform one of the following wsadmin tool options:

- Run scripting commands interactively.

Execute wsadmin with an option other than `-f` or `-c`.

5.1+ An interactive shell appears with a wsadmin prompt. From the wsadmin prompt, enter any JACL or Jython command. You can also invoke commands on the AdminControl, AdminApp, AdminConfig, or Help wsadmin objects.

Using Jacl:

```
wsadmin.sh -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

```

5.1+ Using Jython:

```
Wsadmin.sh -lang jython
```

To leave an interactive scripting session, use the **quit** or **exit** commands. These commands do not take any arguments.

- Run scripting commands as individual commands.

Execute `wsadmin` with the `-c` option.

```
wsadmin.sh -c "\$AdminApp list"
```

or

```
wsadmin.sh -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.

Execute `wsadmin` with the `-f` option, and place the commands you want to execute into the file.

Note: WebSphere for z/OS supports multiple encoding for the `jacl` command files. The default encoding for the `jacl` command file is ASCII. To run an EBCDIC encoded `jacl` file, add the following JVM argument to the `wsadmin.sh` file:

```
-Dscript.encoding=Cp1047
```

You can alternatively have two versions of the `wsadmin.sh` file, one that references the ASCII version of the `jacl` file and another that references the EBCDIC version of the `jacl` file. For example, copy the `wsadmin.sh` file to `wsadminE.sh`. Then add

```
-Dscript.encoding=Cp1047
```

to the `wsadmin.sh` file.

The following examples show how to run scripting commands in a script:

Using `Jacl`:

```
wsadmin.sh -f al.jacl
```

where the `al.jacl` file contains the following commands:

```
set apps [$AdminApp list]
puts $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.

A *profile* is a script that runs before the main script, or before entering interactive mode. You can use profiles to set up a scripting environment customized for the user or the installation.

To run scripting commands in a profile, execute the `wsadmin` tool with the `-profile` option, and place the commands you want to execute into the profile.

For example:

Using `Jacl`:

```
wsadmin.sh -profile alprof.jacl
```

where the `alprof.jacl` file contains the following commands:

```
set apps [$AdminApp list]
puts "Applications currently installed:\n$app"
```

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>
```

5.1+ Using Jython:

```
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
```

5.1+ 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>
```

To customize the script environment, specify one or more profiles to run.

Related concepts

“Scripting objects” on page 41

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

Related reference

“Properties used by scripted administration” on page 237

Specifies the Java properties used by scripting administration.

“Example: Using the wsadmin tool in a secure environment” on page 241

Wsadmin tool

The WebSphere Application Server wsadmin tool provides the ability to execute scripts. You can use the wsadmin tool to manage a WebSphere Application Server V5.0 server installation, as well as configuration, application deployment, and server runtime operations. This tool uses the Bean Scripting Framework (BSF), which supports a variety of scripting languages to configure and control your WebSphere Application Server installation. The WebSphere Application Server only supports the Jacl scripting language.

The command-line invocation syntax for the wsadmin scripting client follows:

```
wsadmin.sh [-h(help)]
```

```
[-?]
```

```
[-c <commands>]
```

```
[-p <properties_file_name>]
```

```
[-profile <profile_script_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 their number, and the `argv` variable contains their contents.

Options

-c command

Designates a single command to execute.

Multiple `-c` options can exist on the command line. They run in the order that you supply them. You must save after each command completes.

-f scriptname

Designates a script to execute.

Only one `-f` option can exist on the command line.

-javaoption

Specifies a valid Java standard or non-standard option.

Multiple `-javaoption` options can exist on the command line.

-lang

Specifies the language of the script file, command, or interactive shell. The possible languages include: Jacl and Jython. The options for the `-lang` argument include: `jacl` and `jython`.

It overrides language determinations that are based on a script file name, or the `com.ibm.ws.scripting.defaultLang` property defined in the `wsadmin.properties` file. There is no default value for the `-lang` argument. If the command line or the property does not supply the script language, and the `wsadmin` tool cannot determine it, an error message generates.

-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 you specify on the command line. The first level is the installation default, `wsadmin.properties`, located in the WebSphere Application Server properties directory. The second level is the user default, `wsadmin.properties`, located in your home directory. The third level is the properties file pointed to by the environment variable `WSADMIN_PROPERTIES`.

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 executes before it invokes this command. If you specify `-f`, the profile executes before it runs the script. In interactive mode, you can use the profile 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.

`-?` 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.

The options SOAP and RMI for `-conntype` also include: host, port, user, and password.

Use the `-conntype NONE` option to run in local mode. The result is that the scripting client is not connected to a running server. This allows for users to manage server configuration and the installation and/or 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:

```
/MyDir/Myjar.jar;/yourdir/yourdir.jar
```

The class path is then added to the class loader for the scripting process.

You can also specify this option in a properties file that is used by the `wsadmin` tool. The property is `com.ibm.ws.scripting.classpath`. If you specify `-wsadmin_classpath` on the command line, the value of this property overrides any value specified in a properties file. The class path property and the command-line options are not concatenated.

In the following syntax examples, *mymachine* is the name of the host in the `wsadmin.properties` file, specified by the `com.ibm.ws.scripting.port` property:

SOAP connection to the local host

Use the options defined in the `wsadmin.properties` file.

SOAP connection to the *mymachine* host

Using Jacl:

```
wsadmin.sh -f test1.jacl -profile setup.jacl  
-conntype SOAP -host mymachine
```

5.1 + Using Jython:

```
wsadmin.sh -lang jython -f test1.py -profile setup.py  
-conntype SOAP -host mymachine
```

Initial and maximum Java heap size

Using Jacl:

```
wsadmin.sh -javaoption -Xms128m -javaoption -Xmx256m -f test.jacl
```

5.1+ Using Jython:

```
wsadmin.sh -lang jython -javaoption -Xms128m  
-javaoption -Xmx256m -f test.py
```

RMI connection with security

Using Jacl:

```
wsadmin.sh -conntype RMI -port rmiportnumber -userid userid  
-password password
```

5.1+ Using Jython:

```
wsadmin.sh -lang jython -conntype RMI -port rmiportnumber  
-userid userid -password password
```

Local mode of operation to perform a single command

Using Jacl:

```
wsadmin.sh -conntype NONE -c "\AdminApp uninstall app"
```

or

```
wsadmin.sh -conntype NONE -c '$AdminApp uninstall app'
```

5.1+ Using Jython:

```
wsadmin.sh -lang jython -conntype NONE -c "\AdminApp.uninstall('app')"
```

or

```
wsadmin.sh -lang jython -conntype NONE -c 'AdminApp.uninstall('app')'
```

Related reference

“wsadmin tool performance tips” on page 243

Jacl

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

Jacl basic commandThe basic syntax for a Jacl command is:

```
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. `puts` writes the string to the I/O stream along with a trailing new line character. 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 `puts` and the other I/O commands. Use `stderr` to identify the standard error output, and use `stdin` to identify the standard input.

Note: When writing Jacl scripts for Windows systems, enclose directory paths that include spaces with quotes. For example:

```
"\Program Files\WebSphere\AppServer\InstallableApps\somesample.ear"
```


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

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

Scripting: Resources for learning

Use the following links to find relevant supplemental information about the Jacl scripting language. 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

- Extending XML tools with Jacl scripts (<http://www-106.ibm.com/developerworks/xml/library/x-xjacl/>)
- Jacl: A Tcl implementation in Java (http://www.usenix.org/publications/library/proceedings/tcl97/full_papers/lam/lam.pdf)
- **5.1+** Charming Jython (<http://www-106.ibm.com/developerworks/java/library/j-jython.html>)
- **5.1+** Sample scripts for WebSphere Application Server (<http://www-106.ibm.com/developerworks/websphere/library/samples/SampleScripts.html>)

Scripting objects

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

WebSphere Application Server System Management separates administrative functions into two categories: functions that deal with the configuration of WebSphere Application Server installations, and functions that deal with the currently running objects in WebSphere Application Server installations.

Scripts deal with both categories of objects. For example, an application server is divided into two distinct entities. One entity represents the configuration of the server which 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 and AdminApp objects 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 the corresponding configuration

supports. The configuration can include many attributes that you cannot query or set from the live running object. The WebSphere Application Server scripting support provides functions to locate configuration objects, and live running objects.

Note: 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 and AdminControl objects, and to obtain interface information about running MBeans.

Help object for scripted administration

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 public commands are available for the Help object:

AdminApp

Provides a summary of all of the available methods for the AdminApp object.

Parameters	none
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help AdminApp
```

5.1+ Using Jython:

```
print Help.AdminApp()
```

Example output: **5.1+**

```
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 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 install 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

AdminConfig

Provides a summary of all of the available methods for the AdminConfig object.

Parameters	none
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help AdminConfig
```

5.1+ 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

AdminControl

Provides a summary of all of the available methods for the AdminControl object.

Parameters	none
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help AdminControl
```

5.1 + 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

all Provides a summary of the information that the MBean defines by name.

Parameters	name -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help all [$AdminControl queryNames type=TraceService,  
process=server1,node=pongo,*]
```

5.1+ 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 setTraceOutputToRingBuffer(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 of the attributes that the MBean defines by name.

Parameters	name -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help attributes [$AdminControl queryNames type=TraceService,
process=server1,node=pongo,*]
```

5.1+ Using Jython:

```
print Help.attributes(AdminControl.queryNames('type=TraceService,
process=server1,node=pongo,*'))
```

Example output:

Attribute Type Access

ringBufferSize java.lang.Integer RW

traceSpecification java.lang.String RW

classname

Provides a class name that the MBean defines by name.

Parameters	name -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help classname [$AdminControl queryNames type=TraceService,
process=server1,node=pongo,*]
```

5.1+ 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 -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help constructors [$AdminControl queryNames type=TraceService,  
process=server1,node=pongo,*]
```

5.1 + 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 -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help description [$AdminControl queryNames type=TraceService,  
process=server1,node=pongo,*]
```

5.1 + 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 of the available methods for the help object.

Parameters	none
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help help
```

5.1 + Using Jython:

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

Example output:

```
WASX7028I: The following functions are supported by Help:
```

```
attributes given an MBean, returns help for attributes
```

```
operations given an MBean, returns help for operations
```

```
given an MBean and an operation name, return signature
```

```
information
```

```
constructors given an MBean, returns help for constructors
```


description given an MBean, returns help for description
notifications given an MBean, returns help for notifications
class name given an MBean, returns help for class name
all given an MBean, returns help for all the above
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

message

Displays information for a message ID.

Parameters	message ID
Return type	java.lang.String

Example usage:

Using Jacl:
\$Help message CNTR0005W

5.1+ Using Jython:

```
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 -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:
\$Help notifications [\$AdminControl queryNames
type=TraceService,process=server1,node=pongo,*]

5.1+ 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 of the operations that the MBean defines by name.

Parameters	name -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help operations [$AdminControl queryNames type=TraceService,
process=server1,node=pongo,*]
```

5.1+ 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 getTraceSpecification(java.lang.String)
void processDumpString(java.lang.String)
void checkTraceString(java.lang.String)
void setTraceOutputToFile(java.lang.String, int, int, java.lang.String)
void setTraceOutputToRingBuffer(int, java.lang.String)
java.lang.String rolloverLogFileImmediate
(java.lang.String, java.lang.String)
```

operations

Provides the signature of the opname operation for the MBean defined by name.

Parameters	name -- java.lang.String, opname -- java.lang.String
Return type	java.lang.String

Example usage:

Using Jacl:

```
$Help operations [$AdminControl queryNames type=TraceService,
process=server1,node=pongo,*] processDumpString
```

5.1+ Using Jython:

```
print Help.operations(AdminControl.queryNames('type=TraceService,
process=server1,node=pongo,*'), 'processDumpString')
```

Example output:

```
void processDumpString(java.lang.String)
```

Description: Write the contents of the Ras services Ring Buffer to the specified file.

Parameters:

Type	java.lang.String
Name	dumpString
Description	a String in the specified format to process or null.

Related concepts

“Jacl” on page 40

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

Related reference

“AdminApp object for scripted administration”

“AdminControl object for scripted administration” on page 75

“AdminConfig object for scripted administration” on page 93

AdminApp object for scripted administration

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 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 will receive a message that you are running in the local mode. If a server is currently running, it is not recommended to run the AdminApp tool in local mode.

The following public commands are available for the AdminApp object:

deleteUserAndGroupEntries

Deletes users or groups for all roles, and deletes userids and passwords for all of the RunAs roles defined in the application.

Parameters:	appname
Return Type:	none

Example usage:

Using Jacl:

```
$AdminApp deleteUserAndGroupEntries myapp
```

5.1+ Using Jython:

```
AdminApp.deleteUserAndGroupEntries('myapp')
```

edit

Edits an application or module in interactive mode.

Parameters using Jacl: appname -- java.lang.String options -- java.lang.String

Parameters using Jython: appname -- java.lang.String; options -- java.lang.String or appname - java.lang.String; options -- java.lang.Object[]

Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminApp edit "JavaMail Sample" {-MapWebModToVH [{"JavaMail
Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}]}
```

5.1 + Using Jython with string options:

```
AdminApp.edit("JavaMail Sample", '[-MapWebModToVH [{"JavaMail
32 Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}]]')
```

5.1 + Using Jython with object options:

```
option = [{"JavaMail 32 Sample WebApp", "mtcomps.war,WEB-INF/web.xml",
"newVH"}] mapVHOption = ["-MapWebModToVH", option]
AdminApp.edit("JavaMail Sample", mapVHOption)
```

Note: The **edit** command changes the application deployment. Specify these changes in the options parameter. No options are required for the **edit** command.

editInteractive

Edits an application or module in interactive mode.

Parameters using Jacl:: appname -- java.lang.String; options -- java.lang.String

Parameters using Jython: appname -- java.lang.String; options -- java.lang.String or appname -- java.lang.String; options -- java.lang.Object[]

Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminApp editInteractive ivtApp
```

5.1 + Using Jython:

```
AdminApp.editInteractive('ivtApp')
```

Note: The **editInteractive** command changes the application deployment. Specify these changes in the options parameter. No options are required for the **editInteractive** command.

export

Exports the application appname parameter to a file you specify by file name.

Parameters: appname, filename

Return Type: none

Example usage:

Using Jacl:

```
$AdminApp export "My App" /usr/me/myapp.ear
```

5.1+ 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
Return Type:	none

Example usage:

Using Jacl:

```
$AdminApp exportDDL "My App" /usr/me/DDL {-ddlprefix myApp}
```

5.1+ Using Jython:

```
AdminApp.exportDDL("My App", '/usr/me/DDL', '[-ddlprefix myApp]')
```

help

Displays general help for the AdminApp object.

Parameters:	none
Return Type:	none

Example usage:

Using Jacl:

```
$AdminApp help
```

5.1+ 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 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 install 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
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp help uninstall
```

5.1+ 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 -- java.lang.String; options -- java.lang.String
Parameters using Jython: earfile -- java.lang.String; options -- java.lang.String or earfile -- java.lang.String; options -- java.lang.Object[]
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp install c:/apps/myapp.ear
```

5.1+ Using Jython:

```
AdminApp.install('c:/apps/myapp.ear')
```

There are many options available for this command. You can obtain a list of valid options for an EAR file with the following command:

Using Jacl:
\$AdminApp options myApp.ear

5.1 + Using Jython:
AdminApp.options('myApp.ear')

You can also obtain help for each object with the following command:

Using Jacl:
\$AdminApp help MapModulesToServers

5.1 + Using Jython:
AdminApp.help('MapModulesToServers')

installInteractive

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 -- java.lang.String; options - java.lang.String
Parameters using Jython:	earfile -- java.lang.String; options -- java.lang.String or earfile -- java.lang.String; options -- java.lang.Object[]
Return Type:	none

Example usage:

Using Jacl:
\$AdminApp installInteractive c:/websphere/
appserver/installableApps/jmsample.ear

5.1 + Using Jython:
AdminApp.installInteractive('c:/websphere/
appserver/installableApps/jmsample.ear')

list

Lists the applications installed in the configuration.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:
\$AdminApp list

5.1 + Using Jython:
print AdminApp.list()

Example output:
adminconsole
DefaultApplication
ivtApp

listModules

Lists the modules in an application.

Parameters: appname -- java.lang.String options -- java.lang.String
Return Type: java.lang.String

The options parameter is optional. The valid option is `-server`. This option lists the application servers on which the modules are installed.

Example usage:

Using Jacl:

```
$AdminApp listModules ivtApp
```

5.1 + 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 EAR file.

Parameters: earfile, options
Return Type: Information about the valid installation options for an EAR file.

Example usage:

Using Jacl:

```
$AdminApp options c:/websphere/appserver/installableApps/jmsample.ear
```

5.1 + Using Jython:

```
AdminApp.options('c:/websphere/appserver/installableApps/jmsample.ear')
```

publishWSDL

Publishes WSDL files for the application specified in the appname parameter to the file specified in the filename parameter.

Parameters: appname, filename
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlerServer c:/temp/a.zip
```

5.1 + Using Jython:

```
AdminApp.publishWSDL('JAXRPCHandlerServer', 'c:/temp/a.zip')
```

publishWSDL

Publishes WSDL files for the application specified in the appname parameter to the file specified in the filename parameter using the soap address prefixes specified in the soapAddressPrefixes parameter.

Parameters: appname, filename, soapAddressPrefixes
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlersServer c:/temp/a.zip  
{{JAXRPCHandlersServerApp.war {{http http://localhost:9080}}}}
```

5.1 + Using Jython:

```
AdminApp.publishWSDL('JAXRPCHandlersServer', 'c:/temp/a.zip',  
'[[JAXRPCHandlersServerApp.war [[http http://localhost:9080]]]]')
```

taskInfo

Provides information about a particular task option for an application file.

Parameters: earfile, task name
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp taskInfo c:/websphere/appserver/  
installableApps/jmsample.ear MapWebModToVH
```

5.1 + 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 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 3 fields: "webModule," "uri," "virtualHost."  
Of these fields, the following may  
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 -- java.lang.String
Return Type: none

Example usage:

Using Jacl:

```
$AdminApp uninstall myApp
```

5.1+ 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.
```

updateAccessIDs

Updates the access id information for users and groups assigned to various roles 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. You should call it after installing an application or after editing security role-specific information for an installed application. This method cannot be invoked when `-conntype` is set to `NONE`. You must be connected to a server to invoke this command.

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.

Parameters:	appname, bALL
Return Type:	none

Example usage:

Using Jacl:

```
$AdminApp updateAccessIDs myapp true
```

5.1+ Using Jython:

```
AdminApp.updateAccessIDs('myapp', 'true')
```

view

View the task specified by the `taskname` option parameter for the application or module 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
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminApp view adminconsole {-tasknames}
```

5.1+ Using Jython:

```
AdminApp.view('adminconsole', ['-tasknames'])
```

Example output:

```
MapModulesToServers
MapWebModToVH
MapRolesToUsers
AppDeploymentOptions
```

Using Jacl:

```
$AdminApp view adminconsole {-AppDeploymentOptions -MapModulesToServers}
```

5.1+ Using Jython:

```
AdminApp.view('Test', ['-AppDeploymentOptions', '-MapModulesToServers'])
```

Example output:

```
MapModulesToServers: Selecting Application Servers
```

Specify the application server where you want to install modules contained in your application. Modules can be installed on the same server or dispersed among several servers:

```
Module: adminconsole
URI: adminconsole.war,WEB-INF/web.xml
Server: WebSphere:cell=juniartiNetwork,node=juniartiManager,server=dmgr
AppDeploymentOptions: Specifying Application Options
```

Specify the various options available to prepare and install your application:

```
Directory to Install Application: $(APP_INSTALL_ROOT)/juniartiNetwork
Distribute Application: Yes
Use Binary Configuration: No
Application Name: adminconsole
Create MBeans for Resources: Yes
Enable Class Reloading: No
Reload Interval in Seconds: 0
```

Example usage:

Using Jacl:

```
$AdminApp view adminconsole#adminconsole.
war+WEB-INF/web.xml {-MapRolesToUsers}
```

5.1+ Using Jython:

```
AdminApp.view('adminconsole#adminconsole.
war+WEB-INF/web.xml', ['-MapRolesToUsers'])
```

Example output:

```
MapRolesToUsers: Mapping Users to Roles
```

Each role defined in the application or module must be mapped to a user or group from the domain's user registry:

```
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:
```

view

View the application or module specified by the name parameter.

Parameters:	name
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminApp view adminconsole
```

5.1+ Using Jython:

```
AdminApp.view('adminconsole')
```

Example output:

```
MapRolesToUsers: Mapping Users to Roles
```

You must map each role defined in the application or module to a user or 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:
MapWebModToVH: Selecting Virtual Hosts for Web Modules
```

Specify the virtual host where you want to install the Web modules contained in your application. You can install Web modules on the same virtual host or dispersed among several hosts:

```
Web Module: adminconsole
URI: adminconsole.war,WEB-INF/web.xml
Virtual Host: admin_host
MapModulesToServers: Selecting Application Servers
```

Specify the application server where you want to install modules contained in your application. You can install modules on the same server or dispersed among several servers.

```
Module: adminconsole
URI: adminconsole.war,WEB-INF/web.xml
Server: WebSphere:cell=juniartiNetwork,node=juniartiManager,server=dmgr
AppDeploymentOptions: Specifying Application Options
```

Specify the various options available to prepare and install your application:

```
Directory to Install Application: $(APP_INSTALL_ROOT)/juniartiNetwork
Distribute Application: Yes
Use Binary Configuration: No
Application Name: adminconsole
Create MBeans for Resources: Yes
Enable Class Reloading: No
Reload Interval in Seconds: 0
```

Related concepts

“Jacl” on page 40

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

Related reference

“Help object for scripted administration” on page 42

“AdminController object for scripted administration” on page 75

“AdminConfig object for scripted administration” on page 93

Installation options for the AdminApp object

You can specify the following installation options for the AdminApp object.

Related reference

“Example: Obtaining information about task options for the AdminApp **install** command” on page 74

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.

BackendIdSelection:

Specifies the backend ID for the enterprise bean 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. This option is not available in an **edit** command.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-BackendIdSelection
{{Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml DB2UDBNT_V72_1}}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-BackendIdSelection  
[[Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml DB2UDBNT_V72_1]]]')
```

BindJndiForEJBMessageBinding:

Binds enterprise beans to listener port names.

Ensure each message-driven enterprise bean in your application or module is bound to a listener port name. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding  
{Ejb1 MessageBean ejb-jar-ic.jar,META-INF/ejb-jar.xml 9999}}
```

5.1+ Using Jython:

```
AdminApp.install(ear, ['-BindJndiForEJBMessageBinding',  
[['Ejb1', 'MessageBean', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 9999]]])
```

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.

BindJndiForEJBNonMessageBinding:

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.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-BindJndiForEJBNonMessageBinding  
{"Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-BindJndiForEJBNonMessageBinding  
[["Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean]]]')
```

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.

cell:

Specifies the cell name for the AdminApp object installation functions.

cluster:

Specifies the cluster name for the AdminApp object installation functions.

Note: This option only applies to a Network Deployment environment.

contextroot:

Specifies the context root you use when installing a stand-alone WAR file.

CorrectOracleIsolationLevel:

Specifies the isolation level for the Oracle type provider. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-CorrectOracleIsolationLevel  
{{AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-CorrectOracleIsolationLevel  
[[AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2]]]')
```

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 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.

CorrectUseSystemIdentity:

Replaces RunAs System to RunAs Roles.

The enterprise beans you install contain RunAs system identity. You can optionally change this identity to a RunAs role. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-CorrectUseSystemIdentity  
{{Inc "Increment Bean Jar"  
Increment.jar,META-INF/ejb-jar.xml getValue() RunAsUser2  
user2 password2} {Inc "Increment  
Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment()  
RunAsUser2 user2 password2}}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-CorrectUseSystemIdentity  
[[Inc "Increment Bean Jar"  
Increment.jar,META-INF/ejb-jar.xml getValue() RunAsUser2  
user2 password2] [Inc "Increment  
Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment()  
RunAsUser2 user2 password2]]]')
```

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.

DataSourceFor10CMPBeans:

Specifies optional data sources for individual 1.x container managed persistence (CMP) beans.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor10CMPBeans
{"Increment Bean Jar" Inc
Increment.jar,META-INF/ejb-jar.xml jdbc/SampleDataSource
user1 password1}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-DataSourceFor10CMPBeans
[["Increment Bean Jar" Inc
Increment.jar,META-INF/ejb-jar.xml jdbc/SampleDataSource
user1 password1]]]')
```

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.

DataSourceFor20CMPBeans:

Specifies optional data sources for individual 2.x container managed persistence (CMP) beans.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor20CMPBeans
{{CustomerEjbJar CustomerEJB
customreEjb.jar,META-INF/ejb-jar.xml ejb/customerEjb
"per connection factory"}
{SupplierEjbJar supplierEjb supplierEjb.jar,META-INF/
ejb-jar.xml ejb/supplierEjb container}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-DataSourceFor20CMPBeans
[[CustomerEjbJar CustomerEJB
customreEjb.jar,META-INF/ejb-jar.xml ejb/customerEjb
```



```
"per connection factory"]
[SupplierEjbJar supplierEjb supplierEjb.jar,META-INF/
ejb-jar.xml ejb/supplierEjb container]]]')
```

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.

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.

DataSourceFor10EJBModules:

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.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor10EJBModules
{"Increment Bean Jar"
Increment.jar,META-INF/ejb-jar.xml jdbc/
SampleDataSource user1 password1}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-DataSourceFor10EJBModules
[["Increment Bean Jar"
Increment.jar,META-INF/ejb-jar.xml jdbc/
SampleDataSource user1 password1]]]')
```

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.

DataSourceFor20EJBModules:

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.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor20EJBModules
{{CustomerEjbJar customreEjb.jar,META-INF/ejb-jar.xml ejb/
customerEjb "per connection factory"
{SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml ejb/
supplierEjb container}}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-DataSourceFor20EJBModules
[[CustomerEjbJar
customreEjb.jar,META-INF/ejb-jar.xml ejb/customerEjb
"per connection factory"]
[SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml
ejb/supplierEjb container]]]')
```

The last field in each entry of this task specifies the value for resource authorization. Valid values for resource authorization is per connection factory or container.

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.

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 datasource.

defaultbinding.datasource.password:

Specifies the password for the default datasource.

defaultbinding.datasource.username:

Specifies the user name for the default datasource.

defaultbinding.ebjndi.prefix:

Specifies the prefix for the enterprise bean Java Naming and Directory Interface (JNDI) name.

defaultbinding.force:

Specifies that the default bindings should 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:

Specifies the location of the properties file for deployment extensions.

deployejb:

Specifies to run EJBDeploy 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 EJBDeploy.

deployejb.dbschema:

Specifies the database schema for EJBDeploy.

deployejb.dbtype:

Specifies the database type for EJBDeploy.

Possible values include the following:

```
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 EJBDeploy.

deployws:

Specifies to deploy WebServices during installation. This option does not require a value.

The default value is: `nodeployws`.

deployws.classpath:

Specifies the extra classpath to use when you deploy WebServices.

deployws.jardirs:

Specifies the extra extension directories to use when you deploy WebServices.

distributeApp:

Specifies that the application management component distributes application binaries. This option does not require a value.

This is the default setting.

EnsureMethodProtectionFor10EJB:

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 update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-EnsureMethodProtectionFor10EJB
{"Increment EJB Module"
 IncrementEJBBean.jar,META-INF/ejb-jar.xml ""}
{"Timeout EJB Module"
 TimeoutEJBBean.jar,META-INF/ejb-jar.xml
 methodProtection.denyAllPermission}}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-EnsureMethodProtectionFor10EJB
[["Increment EJB Module"
 IncrementEJBBean.jar,META-INF/ejb-jar.xml ""] ["Timeout EJB Module"
 TimeoutEJBBean.jar,META-INF/ejb-jar.xml
 methodProtection.denyAllPermission]]]')
```

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.

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.

EnsureMethodProtectionFor20EJB:

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 unchecked. You can assign multiple roles for a method by separating roles names with commas. Use this option to provide missing data or update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-EnsureMethodProtectionFor20EJB
{{CustomerEjbJar customerEjb.jar,META-INF/ejb-jar.xml
 methodProtection.uncheck} {SupplierEjbJar
 supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude}}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-EnsureMethodProtectionFor20EJB
[[CustomerEjbJar 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 separated by commas.

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.

installDir:

Specifies the directory to place the application binaries.

MapModulesToServers:

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 update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapModulesToServers
{"Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml
WebSphere:cell=mycell,node=mynode,server=server1}
{"Default Application" default_app.war,WEB-INF/web.xml
WebSphere:cell=mycell,node=mynode,server=server1}
{"Examples Application" examples.war,WEB-INF/web.xml
WebSphere:cell=mycell,node=mynode,server=server1}}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-MapModulesToServers
[["Increment Bean Jar"
Increment.jar,META-INF/ejb-jar.xml WebSphere:cell=mycell,
node=mynode,server=server1]
["Default Application" default_app.war,WEB-INF/web.xml
WebSphere:cell=mycell,node=mynode,server=server1]
["Examples Application" examples.war,WEB-INF/web.xml
WebSphere:cell=mycell,node=mynode,server=server1]]]')
```

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.

MapEJBRefToEJB:

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.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapEJBRefToEJB
{"Examples Application" ""
examples.war,WEB-INF/web.xml BeenThereBean com.ibm.
websphere.beenthere.BeenThere IncBean}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-MapEJBRefToEJB
[["Examples Application" ""
examples.war,WEB-INF/web.xml BeenThereBean com.ibm.
websphere.beenthere.BeenThere IncBean]]]')
```

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.

MapResEnvRefToRes:

Maps resource environment references to resources. You must map each resource environment reference defined in your application to a resource. Use this option to provide missing data or update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapResEnvRefToRes
{{AsyncSender AsyncSender
asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/
ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-MapResEnvRefToRes
[[AsyncSender AsyncSender
asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/
ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2]]]')
```

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.

MapResRefToEJB:

Maps resource references to resources. You must map each resource reference defined in your application to a resource. Use this option to provide missing data or update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapResRefToEJB {{AsyncSender AsyncSender
asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/MyQueueConnectionFactory
javax.jms.QueueConnectionFactory jms/Resource1}
{"Catalog Component" TheCatalog
catalogEjb.jar,META-INF/ejb-jar.xml jdbc/CatalogDataSource
javax.sql.DataSource jdbc/Resource2}}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-MapResRefToEJB [[AsyncSender AsyncSender
asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/MyQueueConnectionFactory
javax.jms.QueueConnectionFactory jms/Resource1]
["Catalog Component" TheCatalog
catalogEjb.jar,META-INF/ejb-jar.xml jdbc/CatalogDataSource
javax.sql.DataSource jdbc/Resource2]]]')
```

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.

MapRolesToUsers:

Maps users to roles. You must map each role 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 |. Use this option to provide missing data or update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapRolesToUsers [{"All Role" No Yes "" ""}
{"Every Role" Yes No "" ""} {"DenyAllRole No No user1 group1}]}
```

5.1+ Using Jython:

```
AdminApp.install('myapp.ear', '[-MapRolesToUsers [{"All Role" No Yes "" ""}
["Every Role" Yes No "" ""] [DenyAllRole No No user1 group1]]]')
```

where [{"All Role" No Yes "" ""}] corresponds to the following:

"All Role"	is 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

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.

MapRunAsRolesToUsers:

Maps RunAs Roles to users. The enterprise beans you install contain predefined RunAs roles. Enterprise beans that need to run as a particular role for recognition while interacting with another enterprise bean use RunAs roles. Use this option to provide missing data or update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapRunAsRolesToUsers
{{UserRole user1 password1}
{AdminRole administrator administrator}}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-MapRunAsRolesToUsers
[[UserRole user1 password1]
[AdminRole administrator administrator]]]')
```

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.

MapWebModToVH:

Selects virtual hosts for Web modules. Specify the virtual host where you want to install the Web modules 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 update to a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-MapWebModToVH {"Default Application"
default_app.war,WEB-INF/web.xml default_host} {"Examples Application"
examples.war,WEB-INF/web.xml default_host}}}
```

5.1 + Using Jython:

```
AdminApp.install('myapp.ear', '[-MapWebModToVH [{"Default Application"
default_app.war,WEB-INF/web.xml default_host} [{"Examples Application"
examples.war,WEB-INF/web.xml default_host}]]]')
```

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.

noallowPermInFilterPolicy:

Specifies not to continue with the application deployment process when the application contains policy permissions that are in the filter policy. This option does not require a value.

This option is the default setting.

node:

Specifies the node name for the AdminApp installation functions.

nodeployejb:

Specifies not to run EJBDeploy during installation. This option does not require a value.

This value is the default setting.

nodeployws:

Specifies to not deploy WebServices during installation. This option does not require a value.

This value is the default setting.

nodistributeApp:

Specifies that the application management component does not distribute application binaries. This option does not require a value.

The default setting is distributeApp.

nopreCompileJSPs:

Specifies not to precompile JavaServer Pages files. This option does not require a value.

This is the default setting.

nouseMetaDataFromBinary:

Specifies that the metadata used at run time, for example, deployment descriptors, bindings, extensions, and so on, come from the configuration repository. This option does not require a value.

This is the default setting. Use useMetaDataFromBinary to indicate that the metadata used at run time comes from the EAR file.

nousedefaultbindings:

Specifies not to use default bindings for installation. This option does not require a value.

This is the default setting.

preCompileJSPs:

Specifies to precompile JavaServer Pages files. This option does not require a value.

The default is nopreCompileJSPs.

server:

Specifies the server name for the AdminApp installation functions.

update:

Updates the installed application with a new version of the EAR file. This option does not require a value.

The application that is being updated, 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.old:

Specifies that during the update action, bindings from the installed version of the application are ignored. This option does not require a value.

Applies only if you specify the update option.

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.

Applies only if you specify the update option.

useMetaDataFromBinary:

Specifies that the metadata 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 used at run time comes from the configuration repository.

usedefaultbindings:

Specifies to use default bindings for installation. This option does not require a value.

The default setting is `nousedefaultbindings`.

verbose:

Causes additional messages to display during installation. This option does not require a value.

Example: Obtaining information about task options for the AdminApp install command

You can obtain information online about the AdminApp task options by using the AdminApp `taskInfo` method.

- You can use the `options` method to see the requirements for an 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
/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
/mylocation -distributeApp -nouseMetaDataFromBinary}

```

Related reference

“Installation options for the AdminApp object” on page 61

AdminControl object for scripted administration

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: specified by Java Management Extensions (JMX), or 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.

completeObjectName

A convenience command that creates a string representation of a complete ObjectName value based 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
Return Type: java.lang.String

Example usage:

Using Jacl:

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

5.1+ Using Jython:

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

getAttribute

Returns the value of the attribute for the name you provide.

Parameters: name -- java.lang.String; attribute -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=Server,*]
$AdminControl getAttribute $objNameString processType
```

5.1+ Using Jython:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=Server,*')
AdminControl.getAttribute(objNameString, 'processType')
```

getAttribute_jmx

Returns the value of the attribute for the name you provide.

Parameters: name -- ObjectName; attribute -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=Server,*]
set objName [java::new javax.management.ObjectName $objNameString]
$AdminControl getAttribute_jmx $objName processType
```

5.1+ Using Jython:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=Server,*')
import javax.management as mgmt
objName = mgmt.ObjectName(objNameString)
AdminControl.getAttribute_jmx(objName, 'processType')
```

getAttributes

Returns the attribute values for the names 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[]
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=Server,*]
$AdminControl getAttributes $objNameString "cellName nodeName"
```

5.1+ Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=Server,*')
AdminControl.getAttributes(objNameString, ['cellName nodeName'])
```

5.1+ Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=Server,*')
AdminControl.getAttributes(objNameString, ['cellName', 'nodeName'])
```

getAttributes_jmx

Returns the attribute values for the names you provide.

Parameters: name -- ObjectName; attributes -- java.lang.String[]
Return Type: javax.management.AttributeList

Example usage:

Using Jacl:

```
set objectNameString [$AdminControl
completeObjectName WebSphere:type=Server,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attrs [java::new {String[]} 2 {cellName nodeName}]
$AdminControl getAttributes_jmx $objName $attrs
```

5.1+ Using Jython:

```
objectNameString = AdminControl.
completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
attrs = ['cellName', 'nodeName']
AdminControl.getAttributes_jmx(objName, attrs)
```

getCell

Returns the name of the connected cell.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getCell
```

5.1+ Using Jython:

```
AdminControl.getCell()
```

Example output:

```
MyCell
```

getConfigId

A convenience command from an ObjectName or ObjectName fragment that creates a configuration ID. Use this ID with the \$AdminConfig command. Not all MBeans that run have configuration objects that correspond. If there are several MBeans that correspond to an ObjectName fragment, a warning issues and a configuration ID builds for the first MBean it finds.

Parameters:	name -- java.lang.String
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
set serverConfigId [$AdminControl getConfigId
node=SY1,type=Server,*]
```

5.1+ Using Jython:

```
serverConfigId = AdminControl.getConfigId
('node=SY1,type=Server,*')
```

getDefaultDomain

Returns the default domain name from the server.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getDefaultDomain
```

5.1 + Using Jython:

```
AdminControl.getDefaultDomain()
```

Example output:

```
WebSphere
```

getDomainName

Returns the domain name from the server.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getDomainName
```

5.1 + Using Jython:

```
AdminControl.getDomainName()
```

Example output:

```
WebSphere
```

getHost

Returns the name of your host.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getHost
```

5.1 + Using Jython:

```
AdminControl.getHost()
```

Example output:

```
myhost
```

getMBeanCount

Returns the number of MBeans registered in the server.

Parameters:	none
Return Type:	java.lang.Integer

Example usage:

Using Jacl:

```
$AdminControl getMBeanCount
```

5.1+ Using Jython:

```
AdminControl.getMBeanCount()
```

Example output:

```
114
```

getMBeanInfo_jmx

Returns the Java Management Extension MBeanInfo structure that corresponds to an ObjectName value. There is no string signature for this command, because the Help object displays most of the information available from getMBeanInfo.

Parameters:	name -- ObjectName
Return Type:	javax.management.MBeanInfo

Example usage:

Using Jacl:

```
set objectNameString [${AdminControl completeObjectName type=Server,*}]
set objName [${AdminControl makeObjectName $objectNameString}]
${AdminControl getMBeanInfo_jmx $objName}
```

5.1+ Using Jython:

```
objectNameString = AdminControl.completeObjectName('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
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
${AdminControl getNode}
```

5.1+ Using Jython:

```
AdminControl.getNode()
```

Example output:

```
Myhost
```

getPort

Returns the name of your port.

Parameters:	none
Return Type:	java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getPort
```

5.1 + Using Jython:

```
AdminControl.getPort()
```

Example output:

```
8877
```

getPropertiesForDataSource

Deprecated, no replacement.

This command incorrectly assumes the availability of a configuration service when running in connected mode.

Parameters: configId -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]  
$AdminControl getPropertiesForDataSource $ds
```

5.1 + 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.getPropertiesForDataSource(dsArray[0])
```

Example output:

```
WASX7389E: Operation not supported -  
getPropertiesForDataSource command  
is not supported.
```

getType

Returns the connection type.

Parameters: none
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl getType
```

5.1 + Using Jython:

```
AdminControl.getType()
```

Example output:

```
SOAP
```

help

Returns general help text for the AdminControl object.

Parameters: none
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl help
```

5.1+ Using Jython:

```
AdminControl.help()
```

Example output:

```
WASX7027I: The AdminControl object enables the manipulation
           of MBeans running in a WebSphere server
process. The number and type
           of MBeans available to the scripting client
depends 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
connected to this Deployment Manager,
and all the MBeans running in
the application servers on those nodes.
```

The following commands are supported by AdminControl; more detailed information about each of these commands is available by using the "help" command of AdminControl and supplying the name of the command as an argument.

Note that 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.

Hence there is an "invoke" command, as well as a "invoke_jmx" command.

```
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
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl help getAttribute
```

5.1+ 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
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=Server,*]
$AdminControl invoke $objNameString stop
```

5.1+ 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.

Parameters: name -- java.lang.String; operationName -- java.lang.String; params -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=Server,*]
$AdminControl invoke $objNameString appendTraceString
com.ibm.*=all=enabled
```

5.1+ 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

Return Type: 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
```

5.1+ Using Jython:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=Server,*')
AdminControl.invoke(objNameString, 'appendTraceString',
'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[]

Return Type: 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
```

5.1+ 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

Return Type: Boolean

Example usage:

Using Jacl:

```
set objNameString [${AdminControl completeObjectName
WebSphere:type=Server,*}
${AdminControl isRegistered $objNameString
```

5.1+ 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
Return Type: Boolean

Example usage:

Using Jacl:

```
set objectNameString [${AdminControl completeObjectName type=Server,*}
set objName [${AdminControl makeObjectName $objNameString]
${AdminControl isRegistered_jmx $objName
```

5.1+ Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
AdminControl.isRegistered_jmx(objName)
```

makeObjectName

A convenience command that creates an ObjectName value 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 string appends to the name.

Parameters: name -- java.lang.String
Return Type: javax.management.ObjectName

Example usage:

Using Jacl:

```
set objectNameString [${AdminControl completeObjectName
type=Server,node=mynode,*}
set objName [${AdminControl makeObjectName $objNameString]
```

5.1+ Using Jython:

```
objectNameString = AdminControl.completeObjectName
('type=Server,node=mynode,*')
objName = AdminControl.makeObjectName(objectNameString)
```

queryNames

Returns a string that lists all ObjectNames based on the name template.

Parameters: name -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

```
WebSphere:cell=BaseApplicationServerCell,name=server1,  
mbeanIdentifier=server1,type=Server,node=mynode,process=server1
```

queryNames_jmx

Returns a set of ObjectName objects, based on the ObjectName and QueryExp that you provide.

Parameters:	name -- javax.management.ObjectName;query -- javax.management.QueryExp
Return Type:	java.util.Set

Example usage:

Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]  
set objName [$AdminControl makeObjectName $objNameString]  
set null [java::null]  
$AdminControl queryNames_jmx $objName $null
```

5.1+ Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')  
objName = AdminControl.makeObjectName(objectNameString)  
AdminControl.queryNames_jmx(objName, None)
```

Example output:

```
[WebSphere:cell=BaseApplicationServerCell,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
Return Type:	none

Example usage:

Using Jacl:

```
$AdminControl reconnect
```

5.1+ Using Jython:

```
AdminControl.reconnect()
```

Example output:

```
WASX7074I: Reconnect of SOAP connector to host myhost completed.
```

setAttribute

Sets the attribute value for the name you provide.

Parameters: name -- java.lang.String; attributeName -- java.lang.String; attributeValue -- java.lang.String
Return Type: none

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=TraceService,*]
$AdminControl setAttribute $objNameString
traceSpecification com.ibm.*=all=disabled
```

5.1+ Using Jython:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=TraceService,*')
AdminControl.setAttribute(objNameString,
'traceSpecification', 'com.ibm.*=all=disabled')
```

setAttribute_jmx

Sets the attribute value for the name you provide.

Parameters: name -- ObjectName; attribute -- javax.management.Attribute
Return Type: none

Example usage:

Using Jacl:

```
set objectNameString [$AdminControl completeObjectName
WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute
traceSpecification com.ibm.*=all=disabled]
$AdminControl setAttribute_jmx $objName $attr
```

5.1+ 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 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[]
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set objNameString [$AdminControl completeObjectName
WebSphere:type=TracesService,*]
$AdminControl setAttributes $objNameString
{{traceSpecification com.ibm.ws.*=all=enabled}}
```

5.1+ Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=TracesService,*')
AdminControl.setAttributes(objNameString,
['[traceSpecification "com.ibm.ws.*=all=enabled"]'])
```

5.1+ Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectName
('WebSphere:type=TracesService,*')
473 AdminControl.setAttributes(objNameString,
[['traceSpecification', 'com.ibm.ws.*=all=enabled']])
```

setAttributes_jmx

Sets the attribute values for the names you provide and returns a list of successfully set names.

Parameters:	name -- ObjectName; attributes -- javax.management.AttributeList
Return Type:	javax.management.AttributeList

Example usage:

Using Jacl:

```
set objectNameString [$AdminControl completeObjectName
WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute
traceSpecification com.ibm.ws.*=all=enabled]
set alist [java::new javax.management.AttributeList]
$alist add $attr
$AdminControl setAttributes_jmx $objName $alist
```

5.1+ Using Jython:

```
objectNameString = AdminControl.completeObjectName
('WebSphere:type=TraceService,*')
import javax.management as mgmt
objName = AdminControl.makeObjectName(objectNameString)
attr = mgmt.Attribute('traceSpecification',
'com.ibm.ws.*=all=enabled')
alist = mgmt.AttributeList()
alist.add(attr)
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 NodeAgent. It returns a message to indicate if the server starts successfully.

Parameters:	server name -- java.lang.String
Return Type:	java.lang.String

Example usage:

Using Jacl:


```
$AdminControl startServer server1
```

5.1+ 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 NodeAgent. It returns a message to indicate if the server starts successfully.

Parameters: server name -- java.lang.String, wait time -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl startServer server1 100
```

5.1+ 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 NodeAgent or 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
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl startServer server1 myNode
```

5.1+ 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 NodeAgent or 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, wait time -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl startServer server1 myNode 100
```

5.1+ Using Jython:

```
AdminControl.startServer('server1', 'myNode', 100)
```

stopServer

Stops the specified application server. It returns a message to indicate if the server stops successfully.

Parameters: server name -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl stopServer server1
```

5.1+ 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
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl stopServer server1 immediate
```

5.1+ Using Jython:

```
AdminControl.stopServer('server1', 'immediate')
```

stopServer

Stops the specified application server. It returns a message to indicate if the server stops successfully.

Parameters: server name -- java.lang.String, node name -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl stopServer server1 myNode
```

5.1+ 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
Return Type: java.lang.String

Example usage:

Using Jacl:

```
$AdminControl stopServer server1 myNode immediate
```

5.1+ Using Jython:

```
AdminControl.stopServer('server1', 'my Node', 'immediate')
```

testConnection

A convenience command communicates with the DataSourceCfgHelper MBean to test a DataSource connection. This command works with DataSource resided 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. The return value is a message containing the message indicating a successful connection or a connection with warning. If the connection fails, an exception is thrown from the server indicating the error.

Parameters: configId -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]  
$AdminControl testConnection $ds
```

5.1+ 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.
```

TestConnection Deprecated.

This command can give false results and does not work when connected to a NodeAgent. As of V5.0.2, the preferred way to test a Datasource connection is with the **testConnection** command passing in the DataSource configId as the only parameter.

Parameters: configId -- java.lang.String; props -- java.lang.String
Return Type: java.lang.String

Example usage:

Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]
$AdminControl testConnection $ds {{prop1 val1}}
```

5.1+ 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
Return Type:	none

Example usage:

Using Jacl:

```
$AdminControl trace com.ibm.ws.scripting.*=all=enabled
```

5.1+ Using Jython:

```
AdminControl.trace('com.ibm.ws.scripting.*=all=enabled')
```

Related concepts

“Jacl” on page 40

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

Related reference

“AdminApp object for scripted administration” on page 51

“Help object for scripted administration” on page 42

“AdminConfig object for scripted administration” on page 93

Example: Collecting arguments for the AdminControl object

Ensure 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,*]
```

5.1+ Using Jython:

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

- There are multiple ways 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\""
```

5.1+ Using Jython:

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

AdminConfig object for scripted administration

Use the AdminConfig object to invoke configuration commands and to create or change elements of the WebSphere Application Server configuration.

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 will receive a message that you are running in the local mode. If a server is currently running it is not recommended to run the AdminConfig tool in local mode.

The following public commands are available for the AdminConfig object:

attributes

Returns a list of the top level attributes for a given type.

Parameters:

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.

Returns:

a list of attributes

Example usage:

Using Jacl:

```
$AdminConfig attributes ApplicationServer
```

5.1+ 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.

Note: 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 \mydir\myfile $obj
```

5.1+ Using Jython:

```
AdminConfig.checkin('cells/MyCell/Node/MyNode/  
serverindex.xml', '\mydir\myfile', obj)
```

The document URI is relative to the root of the configuration repository, for example, \WebSphere\AppServer\config. The file 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.

convertToCluster

Converts a server so that it is the first member of a new ServerCluster.

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
```

5.1+ 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 -- java.lang.String; parent ID -- java.lang.String; attributes -- java.lang.String
Parameters using Jython: type -- java.lang.String; parent ID -- java.lang.String; attributes -- java.lang.String or type -- java.lang.String; parent ID -- java.lang.String; attributes -- java.lang.Object[]
Returns: a string with configuration object names

Note: The name of the object type that you input here is the one based on the XML configuration files. It 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}}
```

5.1+ Using Jython with string attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')  
AdminConfig.create('DataSource', jdbc1, '[[name ds1]]')
```

5.1+ Using Jython with object attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')
AdminConfig.create('DataSource', jdbc1, [['name', 'dsl']])
```

Example output:

```
dsl(cells/mycell/nodes/DefaultNode/servers/
server1:resources.xml#DataSource_6)
```

createClusterMember

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 specified by the cluster id parameter, and contains attributes specified in the member attributes parameter. The server is created using the server template specified by the template id attribute, and contains the name specified by the memberName attribute. The memberName attribute is required.

Parameters using Jacl:

cluster ID -- java.lang.String; node ID --
java.lang.String; member attributes --
java.lang.String

Parameters using Jython:

cluster ID -- java.lang.String; node ID --
java.lang.String; member attributes --
java.lang.String or cluster ID --
java.lang.String; node ID -- java.lang.String;
member attributes -- java.lang.Object[]

Returns:

the configuration ID of the new cluster
member

Note: The name of the object type that you input here is the one based on the XML configuration files. It 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 createClusterMember $clid $nodeid
{{memberName newMem1} {weight 5}}
```

5.1+ Using Jython with string attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
AdminConfig.createClusterMember(clid, nodeid,
'[[memberName newMem1] [weight 5]]')
```

5.1+ Using Jython with object attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
AdminConfig.createClusterMember(clid, nodeid,
[['memberName', 'newMem1'], ['weight', 5]])
```

Example output:

```
myCluster(cells/mycell/clusters/myCluster:cluster.xml#ClusterMember_2)
```

createDocument

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 \mydir\myfile
```

5.1+ Using Jython:

```
AdminConfig.createDocument('cells/mycell/myfile.xml', '\mydir\myfile')
```

createUsingTemplate

Creates a type of object with the given parent, using a template.

Parameters using Jacl: type -- java.lang.String;
parent id -- java.lang.String;
attributes -- java.lang.String;
template ID --
java.lang.String

Parameters using Jython: type -- java.lang.String;
parent id -- java.lang.String;
attributes -- java.lang.String;
template ID --
java.lang.String or type --
java.lang.String; parent id --
java.lang.String; attributes --
java.lang.Object[]; template
ID -- java.lang.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 createUsingTemplate JDBCProvider
$node {{name newdriver}} $templ
```

5.1+ Using Jython using string attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider',
"DB2 JDBC Provider (XA)")
AdminConfig.createUsingTemplate('JDBCProvider',
node, '['name newdriver]', templ)
```

5.1+ Using Jython using object attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider',
"DB2 JDBC Provider (XA)")
AdminConfig.createUsingTemplate('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

Note: The name of the object type that you input here is the one based on the XML configuration files. It does not have to be the same name that the administrative console displays.

Returns:

a string containing a list of attributes with its type and value

Example usage:

Using Jacl:

```
$AdminConfig defaults TuningParams
```

5.1+ 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	

deleteDocument

Deletes a document from the configuration repository.

The documentURI parameter names the document that will be deleted from the repository.

Parameters:

documentURI

Returns:

none

Example usage:

Using Jacl:

```
$AdminConfig deleteDocument cells/mycell/myfile.xml
```

5.1+ Using Jython:

```
AdminConfig.deleteDocument('cells/mycell/myfile.xml')
```

existsDocument

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
```

5.1+ Using Jython:

```
AdminConfig.existsDocument('cells/mycell/myfile.xml')
```

Example output:

```
1
```

extract

Extracts a configuration repository file described by document URI and places it in the file named by filename.

Note: This method only applies to deployment manager configurations.

Parameters: document URI, filename

Returns: an opaque java.lang.Object to use when checking in the file

Example usage:

Using Jacl:

```
set obj [$AdminConfig extract cells/MyCell/Node/MyNode/  
serverindex.xml \mydir\myfile]
```

5.1+ Using Jython:

```
obj = AdminConfig.extract('cells/MyCell/Node/MyNode/  
serverindex.xml', '\mydir\myfile')
```

The document URI is relative to the root of the configuration repository, for example, \WebSphere\AppServer\config. If the file specified by filename exists, the extracted file replaces it.

getCrossDocumentValidationEnabled

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 containing the message with the cross-document validation setting

Example usage:

Using Jacl:

```
$AdminConfig getCrossDocumentValidationEnabled
```

5.1+ Using Jython:

```
print AdminConfig.getCrossDocumentValidationEnabled()
```

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 described by the given containment path

Example usage:

Using Jacl:

```
$AdminConfig getId /Cell:testcell/Node:testNode/  
JDBCProvider:Db2JdbcDriver/
```

5.1+ 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 there is no corresponding running MBean.

Parameters: configuration id
Returns: a string containing the object name

Example usage:

Using Jacl:

```
set server [$AdminConfig getId /Node:mynode/Server:server1/]  
$AdminConfig getObjectName $server
```

5.1+ Using Jython:

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

Example output:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=cells/mycell/  
nodes/mynode/servers/server1/server.xml#Server_1,type=Server,  
node=mynode,process=server1,processType=UnManagedProcess
```

getSaveMode

Returns the mode used when you invoke a save command.

Possible values include the following:

- overwriteOnConflict - saves changes even if they conflict with other configuration changes
- rollbackOnConflict - causes a save operation to fail if changes conflict with other configuration changes. This value is the default.

Parameters: none
Returns: a string containing the current save mode setting

Example usage:

Using Jacl:

```
$AdminConfig getSaveMode
```

5.1 + Using Jython:

```
print AdminConfig.getSaveMode()
```

Example output:

```
rollbackOnConflict
```

getValidationLevel

Returns the validation used when files are extracted from the repository.

Parameters:

none

Returns:

a string containing the validation level

Example usage:

Using Jacl:

```
$AdminConfig getValidationLevel
```

5.1 + Using Jython:

```
AdminConfig.getValidationLevel()
```

Example output:

```
WASX7189I: Validation level set to HIGH
```

getValidationSeverityResult

Returns the number of validation messages with the given severity from the most recent validation.

Parameters:

severity

Returns:

a string indicating the number of validation messages of the given severity

Example usage:

Using Jacl:

```
$AdminConfig getValidationSeverityResult 1
```

5.1 + Using Jython:

```
AdminConfig.getValidationSeverityResult(1)
```

Example output:

```
16
```

hasChanges

Returns true if unsaved configuration changes exist.

Parameters:

none

Returns:

a string indicating if unsaved configuration changes exist

Example usage:

Using Jacl:
\$AdminConfig hasChanges

5.1 + 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

5.1 + Using Jython:
print AdminConfig.help()

Example output:

```
WASX7053I: The AdminConfig object communicates with the
           Config Service in a WebSphere server to
manipulate configuration data
           for a WebSphere installation. AdminConfig
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 AdminConfig operate in two modes: the default mode is one in which AdminConfig communicates with the WebSphere 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 AdminConfig; more detailed information about each of these commands is available by using the "help" command of AdminConfig and supplying the name of the command as an argument.

attributes Show the attributes for a given type
checkin Check a file into the the config repository.
convertToCluster

converts a server to be the first member of a new ServerCluster

create 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 config 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 attributes of a given type.

deleteDocument
Deletes a document from the config repository.

existsDocument
Tests for the existence of a document in the config repository.

extract
Extract a file from the config repository.

getCrossDocumentValidationEnabled
Returns true if cross-document validation is enabled.

getid
Show the configId of an object, given a string version of its containment

getObjectName
Given a config id, return 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 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
Show help information

list
Lists all configuration objects of a given type

listTemplates
Lists all available configuration templates of a given type.

modify
Change specified attributes of a given configuration object

parents
Show 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
Discard unsaved configuration changes

save
Commit unsaved changes to the configuration repository

setCrossDocumentValidationEnabled
Sets the cross-document validation enabled mode.

setSaveMode
Changes the mode used when "save" is invoked

setValidationLevel
Sets the validation used when files are extracted from the repository.

show
Show the attributes of a given configuration object

showall
Recursively show the attributes of a given configuration object, and all the objects contained within each attribute.

showAttribute
Displays only the value for the single attribute specified.

types
Show the possible types for configuration

validate
Invokes validation

installResourceAdapter

Installs a J2C resource adapter with the given 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:

- rar.name
- rar.desc
- rar.archivePath
- rar.classpath
- rar.nativePath

All options are optional. The rar.name option is the name for the J2CResourceAdapter. If you do not specify this option, the display name in the rar deployment descriptor is used. If that 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 the file is to be extracted. If you do not specify this option, the archive will be extracted to the `/${CONNECTOR_INSTALL_ROOT}` directory. The rar.classpath is the additional class path.

Parameters: rar file name, node, options
Returns: the configuration id of new J2CResourceAdapter object

Example usage:

Using Jacl:

```
$AdminConfig installResourceAdapter /rar/mine.rar  
{-rar.name myResourceAdapter -rar.desc "My rar file"} mynode
```

5.1+ Using Jython:

```
print AdminConfig.installResourceAdapter('/rar/mine.rar',  
    '[-rar.name myResourceAdapter -rar.desc "My rar file"]', 'mynode')
```

Example output:

```
myResourceAdapter(cells/mycell/nodes/mynode:  
resources.xml#J2CResourceAdapter_1)
```

list

Returns a list of objects of a given type, possibly scoped by a parent.

Parameters: 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.

Returns: a list of objects

Example usage:

Using Jacl:

```
$AdminConfig list JDBCProvider
```

5.1+ 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

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.

Returns:

a list of template IDs

Example usage:

Using Jacl:

```
$AdminConfig listTemplates JDBCProvider
```

5.1 + Using Jython:

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

This example displays a list of all JDBCProvider templates available on the system.

modify

Supports modification of object attributes.

Parameters using Jacl:

object -- java.lang.String;
attributes -- java.lang.String

Parameters using Jython:

object -- java.lang.String;
attributes -- java.lang.String
or object -- java.lang.String;
attributes -- java.lang.Object[]

Returns:

none

Example usage:

Using Jacl:

```
$AdminConfig modify ConnFactory1(cells/mycell/nodes/DefaultNode/
servers/deploymentmgr/resources.xml#GenericJMSConnectionFactory_1)
{{userID newID} {password newPW}}
```

5.1 + Using Jython with string attributes:

```
AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/
servers/deploymentmgr/resources.xml#GenericJMSConnectionFactory_1)',
'[[userID newID] [password newPW]]')
```

5.1 + Using Jython with object attributes:

```
AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/
servers/deploymentmgr/resources.xml#GenericJMSConnectionFactory_1)',
[['userID', 'newID'], ['password', 'newPW']])
```


parents

Obtains information about object types.

Parameters:

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.

Returns:

a list of object types

Example usage:

Using Jacl:

```
$AdminConfig parents JDBCProvider
```

5.1+ Using Jython:

```
AdminConfig.parents('JDBCProvider')
```

Example output:

```
Cell  
Node  
Server
```

queryChanges

Returns a list of unsaved configuration files.

Parameters:

none

Returns:

a string containing a list of files with unsaved changes

Example usage:

Using Jacl:

```
$AdminConfig queryChanges
```

5.1+ 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)
```

5.1+ Using Jython:

```
AdminConfig.remove('ds1(cells/mycell/nodes/DefaultNode/
servers/server1:resources.xml#DataSource_6)')
```

required

Displays the required attributes contained by an object of a certain type.

Parameters:

type

Note: The name of the object type that you input here is the one based on the XML configuration files. It does not have to be the same name that the administrative console displays.

Returns:

a string containing a list of required attributes with its type

Example usage:

Using Jacl:

```
$AdminConfig required URLProvider
```

5.1+ 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
```

5.1+ Using Jython:

```
AdminConfig.reset()
```

save

Saves changes in the configuration repository.

Parameters:

none

Returns:

none

Example usage:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

setCrossDocumentValidationEnabled

Sets the cross-document validation enabled mode. Values include true or

false.

Parameters: flag
Returns: none

Example usage:

Using Jacl:

```
$AdminConfig setCrossDocumentValidationEnabled true
```

5.1 + Using Jython:

```
AdminConfig.setCrossDocumentValidationEnabled('true')
```

setSaveMode

Allows you to toggle the behavior of the save command. The default is `rollbackOnConflict`. When a conflict is discovered while saving, the unsaved changes are not committed. The alternative is `overwriteOnConflict` which saves the changes to the configuration repository even if there are conflicts.

Parameters: mode
Returns: none

Example usage:

Using Jacl:

```
$AdminConfig setSaveMode overwriteOnConflict
```

5.1 + Using Jython:

```
AdminConfig.setSaveMode('overwriteOnConflict')
```

setValidationLevel

Sets the validation used when files are extracted from the repository.

There are five validation levels: none, low, medium, high, or highest.

Parameters: level
Returns: a string containing the validation level setting

Example usage:

Using Jacl:

```
$AdminConfig setValidationLevel high
```

5.1 + 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 containing the attribute value

Example usage:

Using Jacl:

```
$AdminConfig show Db2JdbcDriver(cells/mycell/nodes/DefaultNode/resources.xml#JDBCProvider_1)
```

Example output with Jacl:

```
{name "Sample Datasource"} {description  
"Data source for the Sample entity beans"}
```

5.1+ Using Jython:

```
print AdminConfig.show('Db2JdbcDriver  
(cells/mycell/nodes/DefaultNode/resources.xml#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 containing 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 FailingConnectionOnly}  
{reapTime 180}  
{unusedTimeout 1800}}}  
{datasourceHelperClassname com.ibm.websphere.  
rsadapter.CloudscapeDataStoreHelper}  
{description "Datasource for the WebSphere Default Application"}  
{jndiName DefaultDatasource}  
{name "Default Datasource"}  
{propertySet {{resourceProperties {{{description  
"Location of Cloudscape default database."  
{name databaseName}  
{type java.lang.String}  
{value ${WAS_INSTALL_ROOT}/bin/DefaultDB}}  
{{name remoteDataSourceProtocol}  
{type java.lang.String}  
{value {}}}} {{name shutdownDatabase}  
{type java.lang.String}  
{value {}}}} {{name dataSourceName}  
{type java.lang.String}  
{value {}}}} {{name description}  
{type java.lang.String}  
{value {}}}} {{name connectionAttributes}  
{type java.lang.String}  
{value {}}}} {{name createDatabase}  
{type java.lang.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}

```

5.1+ 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 FailingConnectionOnly]
[reapTime 180]
[unusedTimeout 1800]]]
[datasourceHelperClassname com.ibm.websphere.
rsadapter.CloudscapeDataStoreHelper]
[description "Datasource for the WebSphere Default Application"]
[jndiName DefaultDatasource]
[name "Default Datasource"]
[propertySet [[resourceProperties [[description
"Location of Cloudscape default database."]
[name databaseName]
[type java.lang.String]
[value ${WAS_INSTALL_ROOT}/bin/DefaultDB]]
[[name remoteDataSourceProtocol]
[type java.lang.String]
[value []]] [[name shutdownDatabase]
[type java.lang.String]
[value []]] [[name dataSourceName]
[type java.lang.String]
[value []]] [[name description]
[type java.lang.String]
[value []]] [[name connectionAttributes]
[type java.lang.String]
[value []]] [[name createDatabase]
[type java.lang.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]

```

showAttribute

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:	config id, attribute
Returns:	a string containing the attribute value

Example usage:

Using Jacl:

```
set ns [$AdminConfig getid /Node:mynode/]
$AdminConfig showAttribute $ns hostName
```

5.1+ Using Jython:

```
ns = AdminConfig.getid('/Node:mynode/')
print AdminConfig.showAttribute(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
```

5.1+ 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
```

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 containing results of validation

Example usage:

```
Using Jacl:
```

```
$AdminConfig validate
```

5.1+ Using Jython:

```
print AdminConfig.validate()
```

Example output:

```
WASX7193I: Validation results are logged in \WebSphere5\  
AppServer\logs\wsadmin.valout: Total number of messages: 16  
WASX7194I: Number of messages of severity 1: 16
```

Related concepts

“Jacl” on page 40

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

Related reference

“AdminApp object for scripted administration” on page 51

“AdminControl object for scripted administration” on page 75

“Help object for scripted administration” on page 42

ObjectName, Attribute, and AttributeList

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.

WebSphere Application Server `ObjectNames` uniquely identify running objects. `ObjectNames` consist of the following:

- The domain name `WebSphere`.
- Several key properties, for example:
 - **type** - Indicates the type of object that is accessible through the MBean. For example, `ApplicationServer`, `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 `ObjectNames` 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 wild card 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 `ObjectName` 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 `getValue` methods available in the `javax.management.Attribute` class. You can also extract a list of attributes.

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/]
```

5.1+ 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}}}}
```

5.1+ Using Jython list:

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

5.1+ 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
```

5.1+ 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}}
```

5.1+ Using Jython list:

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

5.1+ 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.

Managing configurations with scripting

Configuration management scripts use the AdminConfig object to access the repository where configuration information is stored. You can use the AdminConfig object to list configuration objects and their attributes, create configuration objects, modify configuration objects, remove configuration objects, and obtain help.

1. Decide how you want to execute the script. If you want to run the script immediately from the command line, enter it surrounded by quotes as a parameter to the **wsadmin -c** command. To save the script for repeated use, compose it in a file and execute it with the **wsadmin -f** command. If you want to compose and run the script interactively, issue the **wsadmin** command without the **-c** or **-f** flags. For more information about executing scripts, see *Launching scripting clients*
2. Write an AdminConfig script command statement to perform a management task, for example:

5.1+ Using Jacl:

```
$AdminConfig command
```

5.1+ Using Jython:

```
AdminConfig.command
```

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Creating configuration objects using the wsadmin tool

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 **createFromTemplate** command.

1. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Use the AdminConfig object `listTemplates` command to list available templates:

Using Jacl:

```
$AdminConfig listTemplates JDBCProvider
```

5.1+ 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

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

Using Jacl:

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

5.1+ 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
<i>mynode</i>	is the host name of the node where the new object is added

4. Specify the template that you want to use:

Using Jacl:

```
set t1 [$AdminConfig listTemplates JDBCProvider "DB1 Universal JDBC Driver Provider (XA)"]
```

5.1+ Using Jython:

```
t1 = AdminConfig.listTemplates('JDBCProvider', 'DB1 Universal JDBC Driver 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
<i>DB1 JDBC Provider (XA)</i>	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)*.

5. Create the object with the following command:

Using Jacl:

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

5.1+ 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
<i>newdriver</i>	is the value of the name attribute
t1	evaluates the ID of the template specified in step number 4

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

- Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Specifying configuration objects using the wsadmin tool

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

- Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.

- 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/]
```

5.1+ 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
<i>/type:name/</i>	is the hierarchical containment path of the configuration object
<i>type</i>	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.
<i>name</i>	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:/]
```

5.1+ 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/]
```

5.1+ 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:/]
```

5.1+ 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]
```

5.1+ 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

<i>type</i>	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.
<i>scopeId</i>	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]
```

5.1+ 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]
```

5.1+ Using Jython:

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

3. 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]
```

5.1+ 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

1. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.

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

Using Jacl:

```
$AdminConfig attributes type
```

5.1+ 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
```

5.1+ Using Jython:

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

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

Using Jacl:

```
$AdminConfig required type
```

5.1+ 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
```

5.1+ Using Jython:

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

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

Using Jacl:

`$AdminConfig defaults type`

5.1+ Using Jython:

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

where:

<code>\$</code>	is a Jacl operator for substituting a variable name with its value
<code>AdminConfig</code>	is an object representing the WebSphere Application Server configuration
<code>defaults</code>	is an AdminConfig command
<code>type</code>	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
```

5.1+ Using Jython:

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

Modifying configuration objects with the wsadmin tool

1. Invoke the AdminConfig object commands interactively, in a script, or use `wsadmin -c` from an operating system command prompt.
2. Retrieve the configuration ID of the objects that you want to modify, for example:

Using Jacl:

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

5.1+ Using Jython:

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

where:

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

3. Show the current attribute values of the configuration object with the `show` command, for example:

Using Jacl:

```
$AdminConfig show $jdbcProvider1
```


5.1+ 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

4. Modify the attributes of the configuration object, for example:

Using Jacl:

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

5.1+ Using Jython list:

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

5.1+ 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
<i>This is my new description</i>	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}}
```

5.1+ Using Jython list:

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

5.1+ Using Jython string:

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

5. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Removing configuration objects with the wsadmin tool

Use this task to delete a configuration object from the configuration repository. This action only affects the configuration. 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. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. 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

3. Issue the following command:

Using Jacl:

```
$AdminConfig remove $s1
```

5.1 + 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

4. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1 + 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.

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

For this task, the wsadmin scripting client must be connected to the deployment manager server.

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. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** commands from an operating system command prompt.
2. Set a variable for creating a server:

Using Jacl:

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

5.1 + 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

3. Create a server with the following command:

Using Jacl:

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

5.1 + Using Jython list:

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

5.1 + 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
<i>myserv</i>	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.

- Make a configuration change to the server with the following command:

Using Jacl:

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

5.1+ Using Jython list:

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

5.1+ 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.

- Install an application on the server.
- Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

7. Set the variable for node synchronize.

Note: This step 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:

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

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

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= <i>myNodeName</i>	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 <i>myNodeName</i> , where <i>myNodeName</i> 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.

8. Synchronize by issuing the following command:

Note: This step only applies to network deployment installations.

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

```
true
```

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

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

Configuration management examples with wsadmin

5.1 + There are examples that illustrate how to perform the important configuration management tasks. These examples show how to use features such as templates, and AdminConfig commands. They also show how to configure security, servers, and other resources in your installation. Basic knowledge of the syntax for the Jacl or Jython scripting language is helpful in order to understand and modify the examples.

Example: Finding available templates

Some configuration object types, for example, Server and other resource types, have templates that you can use when you create an object. If you create a JDBCProvider object using the create command, this object is built with the default template. If you know that you need a particular type of JDBCProvider object, you can use a template for that type.

Use the AdminConfig object **listTemplates** command to list available templates, for example:

Using Jacl:

```
$AdminConfig listTemplates JDBCProvider
```

5.1 + Using Jython:

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

- There is a variation of this command that makes it easier to locate a particular template or set of templates. If you supply a string after the name of a type, you only get back a list of templates display names that contain the supplied string, for example:

Using Jacl:

```
$AdminConfig listTemplates JDBCProvider DB2
```

5.1 + Using Jython:

```
AdminConfig.listTemplates('JDBCProvider', 'DB2')
```

This command returns a list of templates with display names containing *DB2*.

- You can use the **show** command with any template like any other configuration object, to see the attributes of the template object, for example:

Using Jacl:

```
set jdbc [$AdminConfig listTemplates JDBCProvider DB2]
set jdbc1 [lindex $jdbc 0]
$AdminConfig show $jdbc1
```

5.1+ Using Jython:

```
jdbc = AdminConfig.listTemplates('JDBCProvider', 'DB2')
# get line separator
import java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayJdbc = jdbc.split(lineSeparator)
jdbc1 = arrayJdbc[0]
AdminConfig.show(jdbc1)
```

Example: Creating new virtual hosts using a template

Some configuration object types have templates that you can use when you create a virtual host.

- Create a new virtual host template *virtualhostname.xml* in the following directory:

```
<WAS-ROOT>\config\templates\custom\
```

- Copy and paste the following file into the new virtual host template:

```
<WAS-ROOT>\config\templates\default\virtualhosts.xml
```

- Edit and customize the new *virtualhostname.xml* file.

- Use the **reset** command to recognize new templates, for example:

Using Jacl:

```
$AdminConfig reset
```

5.1+ Using Jython:

```
AdminConfig.reset()
```

- Use the AdminConfig object **listTemplates** command to list available templates, for example:

Using Jacl:

```
$AdminConfig listTemplates VirtualHost
```

5.1+ Using Jython:

```
print AdminConfig.listTemplates('VirtualHost')
```

Example output:

```
default_host(templates/default:virtualhosts.xml#VirtualHost_1)
my_host(templates/custom:virtualhostname.xml#VirtualHost_1)
```

Note: To list the new templates, restart DeploymentManager or use the AdminConfig object **reset** command.

- Create a new virtual host using the custom template, for example:

Using Jacl:

```
set cell [$AdminConfig getid /Cell:NetworkDeploymentCell/]
set vtempl [$AdminConfig listTemplates VirtualHost my_host]
$AdminConfig createUsingTemplate VirtualHost $cell {{name newVirHost}} $vtempl
```

5.1+ Using Jython:

```
cell = AdminConfig.getid('/Cell:NetworkDeploymentCell/')
vtempl = AdminConfig.listTemplates('VirtualHost', 'my_host')
AdminConfig.createUsingTemplate('VirtualHost', cell, [['name', 'newVirHost']], vtempl)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Note: 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.

Example: Interpreting the output of the AdminConfig attributes command

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"
```

5.1+ Using Jython:

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

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

- Multiple attributes

Using Jacl:

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

5.1+ Using Jython:

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

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

Using Jacl:

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

5.1+ Using Jython:

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

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

- Nested attributes

Using Jacl:

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


5.1+ Using Jython:

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

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

- Attributes that represent lists

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

Using Jacl:

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

5.1+ Using Jython:

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

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

- Reference attributes

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

Using Jacl:

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

5.1+ Using Jython:

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

ExampleType6 objects contain references to ExampleType7 type objects.

- Generic attributes

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

Using Jacl:

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

5.1+ Using Jython:

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

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

(specifying the generic type is possible and legitimate). This is done by specifying `beast:HorseType` instead of `beast`.

Example: Showing attributes with the AdminConfig object

In the wsadmin tool, the AdminConfig **attributes** object displays configuration object types, or object metadata, and does not represent a particular configuration object. This article discusses using the metadata information to show configuration objects.

- Showing simple attributes

Each attribute in a configuration object is represented as a {name value} list.

Using Jacl:

```
$AdminConfig show $myEndPoint {host port}
{host myHost} {port 1234}
```

5.1+ Using Jython:

```
print AdminConfig.show(myEndPoint, ['host', 'port'])
[host myHost] [port 1234]
```

The example configuration object has two attributes. The value of the name attribute is *myHost*. The value of the port attribute is 1234.

- Showing attributes with subtypes

For example, this **show** command returns:

Using Jacl:

```
$AdminConfig show $myex8
{name Halibut} {beast myfish(cells/mycell/adocument.xml#FishType_1)}
```

5.1+ Using Jython:

```
print AdminConfig.show(myex8)
[name Halibut] [beast myfish(cells/mycell/adocument.xml#FishType_1)]
```

The name of the second attribute displays as `beast`. The value of `beast` for this particular `ExampleType8` object has type `FishType`.

- Showing string list attributes

Several attributes on various objects have type `String*`. The `String*` type is a list of strings. These attributes can represent class paths, for example:

Using Jacl:

```
$AdminConfig show $obj1 classpath
{classpath /mine/one.jar;/two.jar;f:/myother/three.jar}
```

5.1+ Using Jython:

```
print AdminConfig.show(obj1, 'classpath')
[classpath c:/mine/one.jar;c:/two.jar;f:/myother/three.jar]
```

Example: Modifying attributes with the AdminConfig object

The **modify** command changes objects in the configuration. In the wsadmin tool, the AdminConfig **attributes** command displays configuration object types, or object metadata, and does not represent a particular configuration object. This article discusses using the metadata information to modify configuration objects.

- Modifying simple attributes

For example:

Using Jacl:

```
$AdminConfig modify $myex1 {{attr1 sampleStringValue}}
```

or

```
$AdminConfig modify $myex1 "{attr1 sampleStringValue}"
```

5.1+ Using Jython:

```
AdminConfig.modify(myex1, [['attr1', 'sampleStringValue']])
```

For multiple attributes:

Using Jacl:

```
$AdminConfig attributes ExampleType2
"attr1 String" "attr2 Boolean" "attr3 Integer"
$AdminConfig modify $myex2 {{attr1 "new string"}
{attr2 false} {attr3 43}}
```

5.1+ Using Jython:

```
print AdminConfig.attributes('ExampleType2')
attr1 String attr2 Boolean attr3 Integer
AdminConfig.modify(myex2, [['attr1', "new string"],
['attr2', 'false'], ['attr3', 43]])
```

In this example, you supply values for all three of the attributes that the ExampleType2 defines. If you enter the following:

Using Jacl:

```
$AdminConfig modify $myex2 {{attr3 43}}
```

5.1+ Using Jython:

```
AdminConfig.modify(myex2, [['attr3', 43]])
```

The attr1 and attr2 attributes would not change.

The following is an example that uses an attribute of ENUM type:

Using Jacl:

```
$AdminConfig attributes ExampleType3
"attr4 ENUM(ALL, SOME, NONE)"
$AdminConfig modify $myex3 {{attr4 NONE}}
```

5.1+ Using Jython:

```
print AdminConfig.attribute('ExampleType3')
attr4 ENUM(ALL, SOME, NONE)
AdminConfig.modify(myex3, [['attr4', 'NONE']])
```

- Modifying nested attributes

For example, you want to modify the ex10 attribute of an object of type ExampleType9:

Using Jacl:

```
$AdminConfig attributes ExampleType9
"attr9a String" "ex10 ExampleType10"
$AdminConfig attributes ExampleType10
"attr10a String"
```

5.1+ Using Jython:

```
print AdminConfig.attributes('ExampleType9')
attr9a String ex10 ExampleType10
AdminConfig.attributes('ExampleType10')
attr10a String
```

There are two ways to modify this object:

- Modify this attribute directly. Perform the following steps:

1. Obtain the configuration ID of the ex10 object:

Using Jacl:

```
$AdminConfig show $myex9
{attr9a MyFavoriteStrings} {ex10 MyEx10(cells/
mycell/adocument.xml#ExampleType10_1)}
```

5.1+ Using Jython:

```
print AdminConfig.attributes('ExampleType9')
[attr9a MyFavoriteStrings] [ex10 MyEx10(cells/
mycell/adocument.xml#ExampleType10_1)]
```

2. Modify the ExampleType10 object:

Using Jacl:

```
$AdminConfig modify MyEx10(cells/mycell/adocument.xml#
ExampleType10_1) {{attr10a yetAnotherString}}
```

5.1+ Using Jython:

```
AdminConfig.modify('MyEx10(cells/mycell/adocument.xml#
ExampleType10_1)', [['attr10a', 'yetAnotherString']])
```

- Modify the nested attribute of its containing object. Use the nested attributes syntax:

Using Jacl:

```
$AdminConfig modify $myex9 {{ex10 {{attr10a yetAnotherString}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(myex9, [['ex10',
[['attr10a', 'yetAnotherString']]])
```

- Modifying list attributes

List attribute values represent a list of objects. For example, the ServerCluster type has an attribute called *BackupClusters* which contains an object collection of the BackupCluster type. The syntax for this type of attribute involves the members of the collection containing an extra set of braces around them, for example:

Using Jacl:

```
$AdminConfig modify $cluster1
{{backupClusters {{{backupClusterName cell20}}
{{backupClusterName bc4}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(cluster1, [['backupClusters
[['backupClusterName', 'cell20']],
[['backupClusterName', 'bc4']]])
```

In this example, the **modify** command takes a string that represents a list of attributes, a single attribute called backupClusters. The backupClusters attribute contains a list of objects of type BackupCluster. The list contains two objects, and each object is represented by exactly one attribute. {{{backupClusterName cell20}}} represents one object of the BackupCluster type, {{{backupClusterName bc4}}} represents a second BackupCluster object, and {{{backupClusterName cell20}}} {{{backupClusterName bc4}}} is the collection of these objects associated with the backupClusters attribute. The same syntax is required even if there is only a single object in the list. When supplying these list attributes, the list is added to the existing list. Be aware that there is no way to delete from a list. To remove the entire list, set the value of a list attribute to an empty string, for example, (""). The **modify** command given in the above example, results in the addition of the backupClusters attribute to the existing contents.

The following is an example of modifying a nested attribute:

Using Jacl:

```
$AdminConfig modify $cluster1
{{name "App Cluster #3"} {preferLocal false} {enableDynamicWLM true}
{backupClusters {{{backupClusterName cell20} {domainBootstrapAddress
{{port 555} {host myhost}}}} {{{backupClusterName rrkbc4}}}}
{members {{{weight 1} {memberName name1}} {{{weight 2}
{memberName name2}} {{{weight 3} {memberName name3}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(cluster1, [['name', "App Cluster #3"],
['preferLocal', 'false'], ['enableDynamicWLM', 'true'], ['backupClusters',
[[['backupClusterName', 'cell20'], ['domainBootstrapAddress',
[['port', 555], ['host', 'myhost']]]], [['backupClusterName', 'rrkbc4']]]],
['members', [[['weight', 1], ['memberName', 'name1']],
[['weight', 2], ['memberName', 'name2']], [['weight', 3],
['memberName', 'name3']]]]])
```

This example list has five attributes: name, preferLocal, enableDynamicWLM, backupClusters, and members. The members attribute is a list of three objects, each of which consists of two sub-attributes, weight and memberName. The backupClusters attribute is a list of two objects. The first of these objects contains one simple subattribute and one complex subattribute. The complex subattribute contains two more subattributes, port and host.

The final variation on specifying configuration objects relates to the fact that some attributes represent generic objects that have different types of values. When you use the AdminConfig attributes command to display the attributes of an object, the various possibilities for specific types are shown in parentheses, for example:

Using Jacl:

```
$AdminConfig attributes MyType
"name String" "components Component(SpecificType1,
SpecificType2, SpecificType3)*"
```

5.1+ Using Jython:

```
print AdminConfig.attributes('MyType')
name String components Component(SpecificType1,
SpecificType2, SpecificType3)*
```

In this example, the component attribute represents a list of objects. Each member of the components attribute is the component type. Component is a generic type with three specific subtypes. When you specify an attribute in a **modify** or **create** command, specify the type of component. If you do not specify the type of component, a generic component object is assumed. To specify a particular component, use components:SpecificType1 instead of components, for example:

Using Jacl:

```
$AdminConfig modify $myMyType
{{name name1} {components:SpecificType2 {{{attr1 val1}
{attr2 val2}} {attr1 val1a} {attr2 val2a}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(myMyType, [['name', 'name1'],
['components:SpecificType2', [['attr1', 'val1'],
['attr2', 'val2']], [['attr1', 'val1a'], ['attr2', 'val2a']]])
```

This command modifies two attributes of the myMyType object: name and components.

- Modifying string list attributes

String list attributes have several attributes on various objects. These objects have the String* type, which is a list of strings. These attributes can represent class paths, for example. To update this list of strings, separate the members of the list by semi-colons. Use semi-colons with all platforms. The syntax for modifying these attributes is a list of strings, and not lists of attribute or name-value pairs, for example:

Using Jacl:

```
$AdminConfig modify $obj1 {{classpath c:/mine/one.jar;
"c:/Program Files/two.jar";f:/myother/three.jar}}
```

5.1+ Using Jython:

```
AdminConfig.modify(obj1, [['classpath', 'c:/mine/one.jar;
"c:/Program Files/two.jar";f:/myother/three.jar']])
```

Example: Listing configuration objects with wsadmin

List configuration objects with the **list** command:

- To get a list of all configuration objects of a specific type, use the following example:

Using Jacl:

```
$AdminConfig list Server
```

5.1+ Using Jython:

```
print AdminConfig.list('Server')
```

Example output:

```
dmgr(cells/mycell/nodes/mynode/servers/dmgr:server.xml#Server_1)
nodeagent(cells/mycell/nodes/mynode/servers/nodeagent:server.xml#Server_1)
server1(cells/mycell/nodes/mynode/servers/server1:server.xml#Server_1)
```

- To list all configuration objects of a given type within a scope:

1. Obtain the ID of the configuration object to be used as the scope.

To use the configuration ID as a scope, the configuration id must be the same for a cell, node, or server object.

To get a configuration ID for a cell object, use the following example:

Using Jacl:

```
set scopeId [$AdminConfig getid /Cell:mycell/]
```

5.1+ Using Jython:

```
scopeId = AdminConfig.getid('/Cell:mycell/')
```

To get a configuration ID for a node object, use the following example:

Using Jacl:

```
set scopeId [$AdminConfig getid /Cell:mycell/Node:mynode/]
```

5.1+ Using Jython:

```
scopeId = AdminConfig.getid('/Cell:mycell/Node:mynode/')
```

To get a configuration ID for a server object, use the following example:

Using Jacl:

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

5.1+ Using Jython:

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

2. List the configuration objects within a scope.

To look for all the server configuration IDs within the node, use one of the following examples:

Using Jacl:

```
set scopeId [$AdminConfig getid /Cell:mycell/Node:mynode/]
$AdminConfig list Server $scopeId
```

```
set scopeId [$AdminConfig getid /Cell:mycell/Node:mynode/]
set nodeServers [$AdminConfig list Server $scopeId]
```

The second example command assigns the returned list to a Jacl variable for subsequent use.

5.1+ Using Jython:

```
scopeId = AdminConfig.getid('/Cell:mycell/Node:mynode/')
AdminConfig.list('Server', scopeId)

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

Example: Identifying valid configuration attributes for objects

You can determine valid attributes by using the AdminConfig **attributes** command.

The following example identifies the valid attributes for a node:

Using Jacl:

```
$AdminConfig attributes Node
```

5.1+ Using Jython:

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

This example produces the following output:

```
"discoveryProtocol ENUM(UDP, TCP, MULTICAST)"
"hostName String"
"name String"
"properties Property(TypedProperty)*"
"shortName String"
```

Example: Changing the location of the activity log

Change the location of the activity log by modifying the serviceLog attribute. The RASLoggingService object contains the serviceLog attribute.

The following example modifies the serviceLog attribute:

Using Jacl:

```
set dmgr1 [$AdminConfig getid /Server:dmgr/]
set rls [$AdminConfig list RASLoggingService $dmgr1]
set logFile [list name \${LOG_ROOT}/newlog.log]
set logAttr [list $logFile]
set attr [list [list serviceLog $logAttr]]
$AdminConfig modify $rls $attr
```

5.1+ Using Jython:

```
dmgr1 = AdminConfig.getid('/Server:dmgr/')
rls = AdminConfig.list('RASLoggingService', dmgr1)
logFile = ['name', '\${LOG_ROOT}/newlog.log']
logAttr = [logFile]
attr = [['serviceLog', logAttr]]
AdminConfig.modify(rls, attr)
```

Example: Modifying port numbers in the serverindex file

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 serverindex.xml file:

- **BOOTSTRAP_ADDRESS**

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 [$AdminConfig getid /Server:server1/]
set ns [$AdminConfig list NameServer $s]
$AdminConfig modify $ns {{BOOTSTRAP_ADDRESS {{port 2810} {host myhost}}}}
```

5.1+ Using Jython:

```
s = AdminConfig.getid('/Server:server1/')
ns = AdminConfig.list('NameServer', s)
AdminConfig.modify(ns, [['BOOTSTRAP_ADDRESS', [['port', 2810], ['host', 'myhost']]]])
```

• **SOAP_CONNECTOR_ADDRESS**

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 [$AdminConfig getid /Server:server1/]
set soap [$AdminConfig list SOAPConnector $s]
$AdminConfig modify $soap {{SOAP_CONNECTOR_ADDRESS {{host myhost} {port 8881}}}}
```

5.1+ Using Jython:

```
s = AdminConfig.getid('/Server:server1/')
soap = AdminConfig.list('SOAPConnector', s)
AdminConfig.modify(soap, [['SOAP_CONNECTOR_ADDRESS', [['host', 'myhost'], ['port', 8881]]])
```

• **DRS_CLIENT_ADDRESS**

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. To modify its end point, obtain the ID of the SystemMessageServer object and issue a **modify** command, for example:

Using Jacl:

```
set s [$AdminConfig getid /Server:server1/]
set sms [$AdminConfig list SystemMessageServer $s]
$AdminConfig modify $sms {{DRS_CLIENT_ADDRESS {{host myhost} {port 7874}}}}
```

5.1+ Using Jython:

```
s = AdminConfig.getid('/Server:server1/')
sms = AdminConfig.list('SystemMessageServer', s)
AdminConfig.modify(sms, [['DRS_CLIENT_ADDRESS', [['host', 'myhost'], ['port', 7874]]])
```

• **JMSERVER_QUEUED_ADDRESS and JMSERVER_DIRECT_ADDRESS**

An attribute of the JMServer 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 JMServer object and issue a **modify** command, for example:

Using Jacl:

```
set s [$AdminConfig getid /Server:server1/]
set jmss [$AdminConfig list JMServer $s]
$AdminConfig modify $jmss {{JMSERVER_QUEUED_ADDRESS {{host myhost} {port 5560}}}}
$AdminConfig modify $jmss {{JMSERVER_DIRECT_ADDRESS {{host myhost} {port 5561}}}}
```

5.1+ Using Jython:

```
s = AdminConfig.getid('/Server:server1/')
jmss = AdminConfig.list('JMServer', s)
AdminConfig.modify(jmss, [['JMSERVER_QUEUED_ADDRESS', [['host', 'myhost'], ['port', 5560]]])
AdminConfig.modify(jmss, [['JMSERVER_DIRECT_ADDRESS', [['host', 'myhost'], ['port', 5561]]])
```

• **NODE_DISCOVERY_ADDRESS**

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 [$AdminConfig getid /Server:server1/]
set nodeAgent [$AdminConfig list NodeAgent $nodeAgentServer]
$AdminConfig modify $nodeAgent {{NODE_DISCOVERY_ADDRESS {{host myhost} {port 7272}}}}
```

5.1+ Using Jython:

```
nodeAgentServer = AdminConfig.getid('/Server:server1/')
nodeAgent = AdminConfig.list('NodeAgent', nodeAgentServer)
AdminConfig.modify(nodeAgent, [['NODE_DISCOVERY_ADDRESS', [['host', 'myhost'],
['port', 7272]]]])
```

- **CELL_DISCOVERY_ADDRESS**

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 [$AdminConfig getid /Server:server1/]
set deploymentManager [$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}}}}
```

5.1+ Using Jython:

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

Example: Disabling a component using wsadmin

An example of disabling the name server component of a configured server follows. You can modify this example to disable a different component.

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

Using Jacl:

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

5.1+ 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)
```

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

Using Jacl:

```
set components [$AdminConfig list Component $server]
```

5.1+ 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)
```

- 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]
```

5.1+ 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)
```

- Disable the name server component by changing the nested initialState attribute belonging to the stateManagement attribute.

Using Jacl:

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

5.1+ Using Jython:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Disabling a service using wsadmin

An example disabling the trace service of a configured server follows. You can modify this example to disable a different service.

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

Using Jacl:

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

5.1+ 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)
```

- List all the services belonging to the server and assign them to the services variable.

Using Jacl:

```
set services [$AdminConfig list Service $server]
```

5.1+ Using Jython:

```
services = AdminConfig.list('Service', server)
print services
```

This command returns a list of 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#
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)
```

- 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 [lindex $services 6]
```

5.1+ Using Jython:

```
# get line separator
import java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayServices = services.split(lineSeparator)
traceService = arrayServices[6]
print traceService
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#TraceService_1)
```

- Disable the trace service by modifying the enable attribute.

Using Jacl:

```
$AdminConfig modify $traceService {{enable false}}
```

5.1+ Using Jython:

```
AdminConfig.modify(traceService, [['enable', 'false']])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a trace using wsadmin

The following example sets the trace for a configured server:

- Identify the server and assign it to the server variable:

Using Jacl:

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

5.1+ 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)
```

- Identify the trace service belonging to the server and assign it to the tc variable:

Using Jacl:

```
set tc [$AdminConfig list TraceService $server]
```

5.1+ Using Jython:

```
tc = AdminConfig.list('TraceService', server)
print tc
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#TraceService_1)
```

- 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}}
```

5.1+ Using Jython:

```
AdminConfig.modify(tc, [['startupTraceSpecification',
'com.ibm.websphere.management.*=all=enabled']])
```

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}}
```

5.1+ Using Jython:

```
AdminConfig.modify(tc, [['startupTraceSpecification',
'com.ibm.websphere.management.*=all=enabled:com.ibm.ws.management.
*=all=enabled:com.ibm.ws.runtime.*=all=enabled']])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring the Java virtual machine using wsadmin

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/]
```

5.1+ 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]
```

5.1+ Using Jython:

```
jvm = AdminConfig.list('JavaVirtualMachine', server1)
print jvm
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:
server.xml#JavaVirtualMachine_1)
(cells/mycell/nodes/mynode/servers/server1:
server.xml#JavaVirtualMachine_2)
```

- Identify the control region JVM of the server and its servant region JVM.

Using Jacl:

```
set cjvm [lindex $jvm 0]
set sjvm [lindex $jvm 1]
```

5.1+ Using Jython:

```
# get line separator
import java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayJVMs = jvm.split(lineSeparator)
cjvm = arrayJVMs[0]
sjvm = arrayJVMs[1]
```

- Modify the JVM to turn on debug.

Using Jacl:

```
$AdminConfig modify $cjvm {{debugMode true} {debugArgs
"-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdpw:transport=
dt_socket,server=y,suspend=n,address=7777"}}
$AdminConfig modify $sjvm {{debugMode true} {debugArgs
"-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdpw:transport=
dt_socket,server=y,suspend=n,address=7777"}}
```

5.1+ Using Jython:

```
AdminConfig.modify(cjvm, [['debugMode', 'true'], ['debugArgs',
"-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdpw:transport=
dt_socket,server=y,suspend=n,address=7777"]])
AdminConfig.modify(sjvm, [['debugMode', 'true'], ['debugArgs',
"-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdpw:transport=
dt_socket,server=y,suspend=n,address=7777"]])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring an enterprise bean container using wsadmin

This topic presents an example of viewing and modifying an enterprise bean container of an application server:

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

Using Jacl:

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

5.1+ 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)
```

- Identify the EJB container belonging to the server and assign it to the ejbContainer variable.

Using Jacl:

```
set ejbContainer [$AdminConfig list EJBContainer $server]
```

5.1+ Using Jython:

```
ejbContainer = AdminConfig.list('EJBContainer', server)
print ejbContainer
```

Example output:

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

- View all the attributes of the EJB container.

The following example command does not show nested attributes:

Using Jacl:

```
$AdminConfig show $ejbContainer
```

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)}
```

5.1+ Using Jython:

```
print AdminConfig.show(ejbContainer)
```

Example output:

```
[cacheSettings (cells/mycell/nodes/myode/servers/
server1:server.xml#EJBCache_1)]
[components []]
[inactivePoolCleanupInterval 30000]
[parentComponent (cells/mycell/nodes/myode/servers/
server1:server.xml#ApplicationServer_1)]
[passivationDirectory ${USER_INSTALL_ROOT}/temp]
[properties []]
[services [(cells/mycell/nodes/myode/servers/
server1:server.xml#MessageListenerService_1)]
[stateManagement (cells/mycell/nodes/mynode/servers/
server1:server.xml#StateManageable_10)]
```

The following command example includes nested attributes:

Using Jacl:

```
$AdminConfig showall $ejbContainer
```

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)}}}
```

5.1+ Using Jython:

```
print AdminConfig.showall(ejbContainer)
```

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)]]]

```

- Modify the attributes.

The following example command modifies the EJB cache settings which are nested attributes:

Using Jacl:

```
$AdminConfig modify $ejbContainer {{cacheSettings
{{cacheSize 2500} {cleanupInterval 3500}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(ejbContainer, [['cacheSettings', [['cacheSize', 2500],
['cleanupInterval', 3500]]]])
```

The following example command modifies the cleanup interval attribute:

Using Jacl:

```
$AdminConfig modify $ejbContainer {{inactivePoolCleanupInterval 15000}}
```

5.1+ Using Jython:

```
AdminConfig.modify(ejbContainer, [['inactivePoolCleanupInterval', 15000]])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring HTTP transport using wsadmin

This example configures the Web container HTTP transport.

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

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

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

- Identify the Web container belonging to the server and assign it to the wc variable.

Using Jacl:

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

5.1+ Using Jython:

```
wc = AdminConfig.list('WebContainer', server)
print wc
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#WebContainer_1)
```

- List all the transports belonging to the Web container and assign it to the transports variable.

Using Jacl:

```
set transportsAttr [$AdminConfig showAttribute $wc transports]
set transports [lindex $transportsAttr 0]
```

5.1+ Using Jython:

```
transportsAttr = AdminConfig.showAttribute(wc, 'transports')
# eliminate the beginning "[" and the end "]"
transports = transportsAttr[1:len(transportsAttr)-1]
```

These commands return the transport objects from the transports attribute in a list format.

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#HTTPTransport_1)
(cells/mycell/nodes/mynode/servers/server1:server.xml#HTTPTransport_2)
```

- Identify the transport to be modified and assign it to the transport variable.

Using Jacl:

```
set transport [lindex $transports 0]
```

5.1+ Using Jython:

```
transport = transports.split(" ")[0]
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#HTTPTransport_1)
```

- Modify the address attribute to change the host and port.

Using Jacl:

```
$AdminConfig modify $transport {{address {{host {myHost}} {port 9081}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(transport, [['address', [['host', 'myHost'], ['port', 9081]]]])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring the message listener service using wsadmin

An example configuring the message listener service for an application server follows:

- Identify the application server and assign it to the server variable:

Using Jacl:

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

5.1+ 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)
```

- Identify the message listener service belonging to the server and assign it to the mls variable:

Using Jacl:

```
set mls [$AdminConfig list MessageListenerService $server]
```

5.1+ Using Jython:

```
m1s = AdminConfig.list('MessageListenerService', server)
print m1s
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:
server.xml#MessageListenerService_1)
```

- Modify various attributes.

This example command changes the thread pool attributes:

Using Jacl:

```
$AdminConfig modify $m1s {{threadPool {{inactivityTimeout 4000}
{isGrowable true} {maximumSize 100} {minimumSize 25}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(m1s, [['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 $m1s listenerPorts]
set lport [lindex $lports 0]
$AdminConfig modify $lport {{maxRetries 2}}
```

5.1+ Using Jython:

```
lports = AdminConfig.showAttribute(m1s, '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 $m1s {{name my}
{destinationJNDIName di} {connectionFactoryJNDIName jndi/fs}}]
$AdminConfig create StateManageable $new {{initialState START}}
```

5.1+ Using Jython:

```
new = AdminConfig.create('ListenerPort', m1s, [['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)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring an ORB service using wsadmin

The following example modifies the Object Request Broker (ORB) service for an application server:

- Identify the application server and assign it to the server variable:

Using Jacl:

```
set server [AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

5.1+ 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)
```

- Identify the ORB belonging to the server and assign it to the orb variable:

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

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

- Modify the attributes:

Using Jacl:

```
$AdminConfig modify $orb {{connectionCacheMaximum 252} {noLocalCopies true}}
```

5.1+ Using Jython:

```
AdminConfig.modify(orb, [['connectionCacheMaximum', 252], ['noLocalCopies', 'true']])
```

This command modifies the connection cache maximum and pass by value attributes. You can modify the example to change the value of other attributes.

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring for database session persistence using wsadmin

The following example configures the session management of a Web container for database session persistence.

Before performing this task you have to create a JDBC provider and create a data source that points to an existing database.

- Identify the application server and assign it to the server variable:

Using Jacl:

```
set server [AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

5.1+ 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)
```

- Identify the session management belonging to the server and assign it to the smgr variable:

Using Jacl:

```
set smgr [$AdminConfig list SessionManager $server]
```

5.1+ Using Jython:

```
smgr = AdminConfig.list('SessionManager', server)
print smgr
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#SessionManager_1)
```

- Modify database session persistence:

Using Jacl:

```
$AdminConfig modify $smgr {{sessionDatabasePersistence
{{datasourceJNDIName jdbc/mySession} {userId myUser}
{password myPassword}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['sessionDatabasePersistence',
[['datasourceJNDIName', 'jdbc/mySession'],
['userId', 'myUser'], ['password', 'myPassword']]])
```

This command sets the minimum set of attributes to configure database session persistence. You can optionally modify the db2RowSize and tableSpaceName attributes too.

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring for serialization session access using wsadmin

The following example configures session management of a Web container for serialization session access.

- Identify the application server and assign it to the server variable:

Using Jacl:

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

5.1+ 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)
```

- Identify the session management belonging to the server and assign it to the smgr variable:

Using Jacl:

```
set smgr [$AdminConfig list SessionManager $server]
```

5.1+ Using Jython:

```
smgr = AdminConfig.list('SessionManager', server)
print smgr
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#SessionManager_1)
```

- Enable serialization session access.

- The following example sets the maximum wait time a servlet waits on a session before continuing processing:

Using Jacl:

```
$AdminConfig modify $smgr {{allowSerializedSessionAccess true}
{maxWaitTime 20}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['allowSerializedSessionAccess', 'true'],
['maxWaitTime', 20]])
```

- In the following example, servlet processing can stop when the session request times out:

Using Jacl:

```
$AdminConfig modify $smgr {{allowSerializedSessionAccess true}
{maxWaitTime 20} {accessSessionOnTimeout true}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['allowSerializedSessionAccess', 'true'],
['maxWaitTime', 20], ['accessSessionOnTimeout', 'true']])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring for session tracking using wsadmin

The following example configures the session management of a Web container for session tracking:

- Identify the application server and assign it to the server variable:

Using Jacl:

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

5.1+ 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)
```

- Identify the session management belonging to the server and assign it to the smgr variable:

Using Jacl:

```
set smgr [$AdminConfig list SessionManager $server]
```

5.1+ Using Jython:

```
smgr = AdminConfig.list('SessionManager', server)
print smgr
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#SessionManager_1)
```

- Modify attributes related to session tracking:

- This example command enables cookies and modifies cookie setting:

Using Jacl:

```
$AdminConfig modify $smgr {{enableCookies true}
{defaultCookieSettings {{maximumAge 10}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['enableCookies', 'true'],
['defaultCookieSettings', [['maximumAge', 10]]]])
```

- This example command enables protocol switch rewriting:

Using Jacl:

```
$AdminConfig modify $smgr {{enableProtocolSwitchRewriting true}
{enableUrlRewriting false} {enableSSLTracking false}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['enableProtocolSwitchRewriting', 'true'],
['enableUrlRewriting', 'false'], ['enableSSLTracking', 'false']])
```

- This example command enables URL rewriting:

Using Jacl:

```
$AdminConfig modify $smgr {{enableUrlRewriting true}
{enableProtocolSwitchRewriting false} {enableSSLTracking false}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['enableUrlRewriting', 'true'],
['enableProtocolSwitchRewriting', 'false'], ['enableSSLTracking', 'false']])
```

- This example command enables SSL tracking:

Using Jacl:

```
$AdminConfig modify $smgr {{enableSSLTracking true}
{enableProtocolSwitchRewriting false} {enableUrlRewriting false}}
```

5.1+ Using Jython:

```
AdminConfig.modify(smgr, [['enableSSLTracking', 'true'],
['enableProtocolSwitchRewriting', 'false'],
['enableUrlRewriting', 'false']])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring for processes using wsadmin

The following example modifies the process definition of a server:

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

Using Jacl:

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

5.1+ 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 process definition belonging to this server and assign it to the processDef variable:

Using Jacl:

```
set processDef [$AdminConfig list JavaProcessDef $server1]
set controllerProcessDef [lindex $processDefs 1]
set servantProcessDef [lindex $processDefs 1]
```

5.1+ Using Jython:

```
processDef = AdminConfig.list('JavaProcessDef', server1)
# get line separator
import java
```

```

lineSeparator = java.lang.System.getProperty('line.separator')
arrayPDs = processDefs.split(lineSeparator)
controllerProcessDef = arrayPDs[0]
servantProcessDef = arrayPDs[1]
print controllerProcessDef
print servantProcessDef

```

Example output:

```

(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaProcessDef_1)
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaProcessDef_2)

```

- Change the attributes.
 - The following examples show how to change the process definition of the servant region. You can change the process definition of the controller region by substituting controllerProcessDef for servantProcessDef .

Using Jacl:

```
$AdminConfig modify $servantProcessDef {{workingDirectory /temp/user1}}
```

5.1+ Using Jython:

```
AdminConfig.modify(servantProcessDef, [['workingDirectory', '/temp/user1']])
```

- This example modifies the stderr file name:

Using Jacl:

```

set errFile [list stderrFilename \${LOG_ROOT}/server1/new_stderr.log]
set attr [list $errFile]
$AdminConfig modify $servantProcessDef [subst {{ioRedirect ${attr}}}]

```

5.1+ Using Jython:

```

errFile = ['stderrFilename', '${LOG_ROOT}/server1/new_stderr.log']
attr = [errFile]
AdminConfig.modify(servantProcessDef, [['ioRedirect', attr]])

```

- This example modifies the process priority:

Using Jacl:

```
$AdminConfig modify $servantProcessDef {{execution {{processPriority 15}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(servantProcessDef, [['execution', [['processPriority', 15]]]])
```

- This example changes the maximum startup attempts:

Using Jacl:

```
$AdminConfig modify $servantProcessDef {{monitoringPolicy {{maximumStartupAttempts 1}}}}
```

5.1+ Using Jython:

```
AdminConfig.modify(servantProcessDef, [['monitoringPolicy', [['maximumStartupAttempts', 1]]]])
```

You can modify this example to change other attributes in the process definition object.

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a shared library using wsadmin

The following example configures an application server to use a shared library.

- Identify the server and assign it to the server variable:

Using Jacl:

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

5.1+ 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)
```

- Create the shared library in the server:

Using Jacl:

```
$AdminConfig create Library $server {{name mySharedLibrary}
{classPath /mySharedLibraryClasspath}}
```

5.1+ Using Jython:

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

Example output:

```
MysharedLibrary(cells/mycell/nodes/mynode/servers/
server1:libraries.xml#Library_1)
```

- Identify the application server from the server and assign it to the appServer variable:

Using Jacl:

```
set appServer [$AdminConfig list ApplicationServer $server]
```

5.1+ Using Jython:

```
appServer = AdminConfig.list('ApplicationServer', server)
print appServer
```

Example output:

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

- Identify the class loader in the application server and assign it to the classLoader variable.

To use the existing class loader associated with the server, the following commands use the first class loader:

Using Jacl:

```
set classLoaders [$AdminConfig showAttribute $appServer classloaders]
set classLoader [lindex $classLoaders 0]
```

5.1+ Using Jython:

```
classLoaders = AdminConfig.showAttribute(appServer, 'classloaders')
cleanClassLoaders = classLoaders[1:len(classLoaders)-1]
classLoader = cleanClassLoaders.split(' ')[0]
```

Create a new class loader, by doing the following:

Using Jacl:

```
set classLoader [$AdminConfig create Classloader
$appServer {{mode PARENT_FIRST}}]
```

5.1+ Using Jython:

```
classLoader = AdminConfig.create('Classloader',
appServer, [['mode', 'PARENT_FIRST']])
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#Classloader_1)
```

- Associate the created shared library with the application server through the class loader.

Using Jacl:

```
$AdminConfig create LibraryRef $classLoader {{libraryName MyshareLibrary}
{sharedClassLoader true}}
```

5.1+ Using Jython:

```
print AdminConfig.create('LibraryRef', classLoader,
[['libraryName', 'MyshareLibrary'], ['sharedClassLoader', 'true']])
```

Example output:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a variable map using wsadmin

This example creates a variable map to an existing server.

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

Using Jacl:

```
set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server2/]
```

5.1+ Using Jython:

```
server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server2/')
print server
```

Example output:

```
server2(cells/mycell/nodes/mynode/servers/server2:server.xml#Server_2)
```

- Create an empty variable map for the server and assign it to the varMap variable.

Using Jacl:

```
set varMap [$AdminConfig create VariableMap $server {}]
```

5.1+ Using Jython:

```
varMap = AdminConfig.create('VariableMap', server, [])
print varMap
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server2:variables.xml#VariableMap_1)
```

This example is to create a variable map. If your server already has an existing variable map, then you can use the following to get its configuration object:

Using Jacl:

```
set varMap [$AdminConfig getid /Cell:mycell/Node:mynode/
Server:server2/VariableMap:/]
```

5.1+ Using Jython:

```
varMap = AdminConfig.getid('/Cell:mycell/Node:mynode/
Server:server2/VariableMap:/')
```

- Set up variable map entry attributes.

In the following example, you create variable map entries DB2_INSTALL_ROOT and DB2_LIB_DIR. DB2_LIB_DIR is going to refer back to DB2_INSTALL_ROOT:

Using Jacl:

```
set nameattr1 [list symbolicName DB2_INSTALL_ROOT]
set valattr1 [list value "/opt/IBM/db2/sqllib"]
set nameattr2 [list symbolicName DB2_LIB_DIR]
```



```

set valattr2 [list value "${DB2_INSTALL_ROOT}/lib"]
set attr1 [list $nameattr1 $valattr1]
set attr2 [list $nameattr2 $valattr2]
set attrs [list $attr1 $attr2]

```

Example output:

```

{{symbolicName DB2_INSTALL_ROOT} {value /opt/IBM/db2/sqllib}}
{{symbolicName DB2_LIB_DIR} {value ${DB2_INSTALL_ROOT}/lib}}

```

Using Jython:

```

nameattr1 = ['symbolicName', 'DB2_INSTALL_ROOT']
valattr1 = ['value', "/opt/IBM/db2/sqllib"]
nameattr2 = ['symbolicName', 'DB2_LIB_DIR']
valattr2 = ['value', "\${DB2_INSTALL_ROOT}/lib"]
attr1 = [nameattr1, valattr1]
attr2 = [nameattr2, valattr2]
attrs = [attr1, attr2]
print attrs

```

Example output:

```

[[symbolicName, DB2_INSTALL_ROOT], [value, c:/db2/sqllib
/opt/IBM/db2/sqllib]], [[symbolicName, DB2_LIB_DIR],
[value, "\${DB2_INSTALL_ROOT}/lib"]]

```

- Modify the entries attribute in the variable map to add the two new entries.

Using Jacl:

```
$AdminConfig modify $varMap [subst {{entries {$attrs}}}]
```

5.1+ Using Jython:

```
AdminConfig.modify(varMap, [['entries', attrs]])
```

- To view the variable map:

Using Jacl:

```
$AdminConfig showall $varMap
```

Example output:

```

{entries {{{symbolicName DB2_INSTALL_ROOT}
{value c:/db2/sqllib}} {{symbolicName DB2_LIB_DIR}
{value ${DB2_INSTALL_ROOT}/lib}}}}

```

5.1+ Using Jython:

```
print AdminConfig.showall(varMap)
```

Example output:

```

[entries [[[symbolicName DB2_INSTALL_ROOT]
[value c:/db2/sqllib]] [[symbolicName DB2_LIB_DIR]
[value ${DB2_INSTALL_ROOT}/lib]]]]

```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring name space bindings using wsadmin

This example configures name space binding on a cell.

- Identify the cell and assign it to the cell variable.

Using Jacl:

```
set cell [$AdminConfig getid /Cell:mycell/]
```

Example output:

```
mycell(cells/mycell/cell.xml#Cell_1)
```

5.1+ Using Jython:

```
cell = AdminConfig.getid('/Cell:mycell/')
print cell
```

You can change this example to configure on a node or server here.

- Add a new name space binding on the cell. There are four binding types to choose from when configuring a new name space binding. They are string, EJB, CORBA, and indirect.
 - To configure a string type name space binding:

Using Jacl:

```
$AdminConfig create StringNameSpaceBinding $cell {{name binding1}
{nameInNameSpace myBindings/myString} {stringToBind
"This is the String value that gets bound"}}
```

Example output:

```
binding1(cells/mycell:namebindings.xml#StringNameSpaceBinding_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('StringNameSpaceBinding', cell,
[['name', 'binding1'], ['nameInNameSpace', 'myBindings/myString'],
['stringToBind', "This is the String value that gets bound"]])
```

- To configure an EJB type name space binding:

Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2}
{nameInNameSpace myBindings/myEJB} {applicationNodeName mynode}
{bindingLocation SINGLESERVER} {applicationServerName server1}
{ejbJndiName ejb/myEJB}}
```

5.1+ Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name', 'binding2'],
['nameInNameSpace', 'myBindings/myEJB'], ['applicationNodeName', 'mynode'],
['bindingLocation', 'SINGLESERVER'], ['applicationServerName', 'server1'],
['ejbJndiName', 'ejb/myEJB']])
```

This example is for an enterprise bean located in a server. For an enterprise bean in a cluster, change the configuration example to:

Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2}
{nameInNameSpace myBindings/myEJB} {bindingLocation SERVERCLUSTER}
{applicationServerName cluster1} {ejbJndiName ejb/myEJB}}
```

5.1+ Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name', 'binding2'],
['nameInNameSpace', 'myBindings/myEJB'], ['bindingLocation', 'SERVERCLUSTER'],
['applicationServerName', 'cluster1'], ['ejbJndiName', 'ejb/myEJB']])
```

Example output:

```
binding2(cells/mycell:namebindings.xml#EjbNameSpaceBinding_1)
```

- To configure a CORBA type name space binding:

Using Jacl:

```
$AdminConfig create CORBAObjectNameSpaceBinding $cell {{name binding3}
{nameInNameSpace myBindings/myCORBA} {corbanameUrl
corbaname:iiop:somehost.somecompany.com:2809#stuff/MyCORBAObject}
{federatedContext false}}
```

Example output:

```
binding3(cells/mycell:namebindings.xml#CORBAObjectNameSpaceBinding_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('CORBAObjectNamespaceBinding', cell,
  [['name', 'binding3'], ['nameInNameSpace', 'myBindings/myCORBA'],
  ['corbanameUrl', 'corbaname:iiop:somehost.somecompany.com:2809#stuff/
  MyCORBAObject'], ['federatedContext', 'false']])
```

- To configure an indirect type name space binding:

Using Jacl:

```
$AdminConfig create IndirectLookupNameSpaceBinding $cell {{name binding4}
{nameInNameSpace myBindings/myIndirect} {providerURL corbaloc::
myCompany.com:9809/NameServiceServerRoot} {jndiName jndi/name/for/EJB}}
```

Example output:

```
binding4(cells/mycell:namebindings.xml#IndirectLookupNameSpaceBinding_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('IndirectLookupNameSpaceBinding', cell,
  [['name', 'binding4'], ['nameInNameSpace', 'myBindings/myIndirect'],
  ['providerURL', 'corbaloc::myCompany.com:9809/NameServiceServerRoot'],
  ['jndiName', 'jndi/name/for/EJB']])
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Creating a cluster using wsadmin

An example creating a cluster using an existing server follows:

- 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/]
```

5.1+ Using Jython:

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

- 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.

5.1+ Using Jython:

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

An example of this output follows:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Creating a cluster member using wsadmin

An example creating a cluster member to an existing cluster follows:

- Identify the existing cluster and assign it to the cluster variable:

Using Jacl:

```
set cluster [$AdminConfig getid /ServerCluster:myCluster1/]
```

5.1+ Using Jython:

```
cluster = AdminConfig.getid('/ServerCluster:myCluster1/')
print cluster
```

An example of this output follows:

```
myCluster1(cells/mycell/cluster/myCluster1:cluster.xml#ServerCluster_1)
```

- Identify the node to create the new server and assign it to the node variable:

Using Jacl:

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

5.1+ Using Jython:

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

An example of this output follows:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

- (Optional) Identify the cluster member template and assign it to the serverTemplate variable:

Using Jacl:

```
set serverTemplate [$AdminConfig listTemplates Server]
```

5.1+ Using Jython:

```
serverTemplate = AdminConfig.listTemplates('Server')
print serverTemplate
```

An example of this output follows:

```
server1(templates/default/nodes/servers/server1:server.xml#Server_1)
```

- 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}}
```

5.1+ 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
```

5.1+ Using Jython:

```
print AdminConfig.createClusterMember(cluster, node,
[['memberName', 'clusterMember1']], serverTemplate)
```

An example of this output follows:

```
clusterMember1(cells/mycell/clusters/
myCluster1:cluster.xml$ClusterMember_2)
```

Example: Configuring a JDBC provider using wsadmin

An example configuring a new JDBC provider follows:

- Identify the parent ID and assign it to the node variable.

Using Jacl:

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

This example uses the node configuration object as the parent. You can modify this example to use cell or server configuration object as the parent.

5.1+ Using Jython:

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

An example of this output follows:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required JDBCProvider
```

5.1+ Using Jython:

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

An example of this output follows:

```
Attribute      Type
name           String
implementationClassName  String
```

- Set up the required attributes and assign it to the jdbcAttrs variable:

Using Jacl:

```
set n1 [list name JDBC1]
set implCN [list implementationClassName myclass]
set jdbcAttrs [list $n1 $implCN]
```

An example of this output follows:

```
{name {JDBC1}} {implementationClassName {myclass}}
```

You can modify the example to setup non-required attributes for JDBC provider.

5.1+ Using Jython:

```
n1 = ['name', 'JDBC1']
implCN = ['implementationClassName', 'myclass']
jdbcAttrs = [n1, implCN]
print jdbcAttrs
```

5.1+ An example of this output follows:

```
[['name', 'JDBC1'], ['implementationClassName', 'myclass']]
```

- Create a new JDBC provider using node as the parent:

Using Jacl:

```
$AdminConfig create JDBCProvider $node $jdbcAttrs
```

5.1+ Using Jython:

```
AdminConfig.create('JDBCProvider', node, jdbcAttrs)
```

An example of this output follows:

```
JDBC1(cells/mycell/nodes/mynode:resources.xml#JDBCProvider_1)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new data source using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjdbc [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/]
```

5.1+ Using Jython:

```
newjdbc = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/')
print newjdbc
```

Example output:

```
JDBC1(cells/mycell/nodes/mynode:resources.xml#JDBCProvider_1)
```

- Obtain the required attributes:

Using Jacl:

```
$AdminConfig required DataSource
```

5.1+ Using Jython:

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

Example output:

```
Attribute Type
name String
```

- Setting up required attributes:

Using Jacl:

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

5.1+ Using Jython:

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

- Create a data source:

Using Jacl:

```
set newds [$AdminConfig create DataSource $newjdbc $dsAttrs]
```

5.1+ Using Jython:

```
newds = AdminConfig.create('DataSource', newjdbc, dsAttrs)
print newds
```

Example output:

```
DS1(cells/mycell/nodes/mynode:resources.xml#DataSource_1)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new connection pool using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
```

5.1+ Using Jython:

```
newds = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/')

```

Example output:

```
DS1(cells/mycell/nodes/mynode:resources.xml$DataSource_1)
```

- Creating connection pool:

Using Jacl:

```
$AdminConfig create ConnectionPool $newds {}
```

5.1+ Using Jython:

```
print AdminConfig.create('ConnectionPool', newds, [])
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#ConnectionPool_1)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new data source custom property using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
```

5.1+ 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)
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newds propertySet]
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newds, 'propertySet')  
print propSet
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#J2EEResourcePropertySet_8)
```

- Get required attribute:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

Example output:

```
Attribute      Type  
name           String
```

- Set up attributes:

Using Jacl:

```
set name [list name RP4]  
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP4']  
rpAttrs = [name]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

Example output:

```
RP4(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_8)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new J2CAuthentication data entry using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set security [$AdminConfig getid /Cell:mycell/Security:/]
```

Example output:

```
(cells/mycell:security.xml#Security_1)
```

5.1+ Using Jython:

```
security = AdminConfig.getid('/Cell:mycell/Security:/')
print security
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required JAASAuthData
```

Example output:

Attribute	Type
alias	String
userId	String
password	String

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
alias = ['alias', 'myAlias']
userid = ['userid', 'myid']
password = ['password', 'secret']
jaasAttrs = [alias, userid, password]
print jaasAttrs
```

5.1+ Example output:

```
[['alias', 'myAlias'], ['userid', 'myid'], ['password', 'secret']]
```

- Create JAAS auth data:

Using Jacl:

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

Example output:

```
(cells/mycell:security.xml#JAASAuthData_2)
```

5.1+ Using Jython:


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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WAS40 data source using wsadmin:

- Identify the parent ID:

Using Jacl:

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

Example output:

```
JDBC1(cells/mycell/nodes/mynode:resources.xml$JDBCProvider_1)
```

5.1+ Using Jython:

```
newjdbc = AdminConfig.getid('/JDBCProvider:Cloudscape JDBC Provider/')  
print newjdbc
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required WAS40DataSource
```

Example output:

```
Attribute  Type  
name      String
```

5.1+ Using Jython:

```
print AdminConfig.required('WAS40DataSource')
```

- Set up required attributes:

Using Jacl:

```
set name [list name was4DS1]  
set ds4Attrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'was4DS1']  
ds4Attrs = [name]
```

- Create WAS40DataSource:

Using Jacl:

```
set new40ds [$AdminConfig create WAS40DataSource $newjdbc $ds4Attrs]
```

Example output:

```
was4DS1(cells/mycell/nodes/mynode:resources.xml#WAS40DataSource_1)
```

5.1+ Using Jython:

```
new40ds = AdminConfig.create('WAS40DataSource', newjdbc, ds4Attrs)  
print new40ds
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WAS40 connection pool using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/Server:
server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
```

Example output:

```
was4DS1(cells/mycell/nodes/mynodes:resources.xml$WAS40DataSource_1)
```

5.1+ Using Jython:

```
new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:
server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')
print new40ds
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required WAS40ConnectionPool
```

Example output:

Attribute	Type
minimumPoolSize	Integer
maximumPoolSize	Integer
connectionTimeout	Integer
idleTimeout	Integer
orphanTimeout	Integer
statementCacheSize	Integer

5.1+ Using Jython:

```
print AdminConfig.required('WAS40ConnectionPool')
```

- 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 cpAttrs [list $minps $maxps $conn $idle $orphan $scs]
```

Example output:

```
{minimumPoolSize 5} {maximumPoolSize 30}
{connectionTimeout 10} {idleTimeout 5}
{orphanTimeout 5} {statementCacheSize 5}
```

5.1+ 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
```

5.1+ Example output:

```
[[minimumPoolSize, 5], [maximumPoolSize, 30],
[connectionTimeout, 10], [idleTimeout, 5],
[orphanTimeout, 5], [statementCacheSize, 5]]
```

- Create was40 connection pool:

Using Jacl:

```
$AdminConfig create WAS40ConnectionPool $new40ds $40cpAttrs
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#WAS40ConnectionPool_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('WAS40ConnectionPool', new40ds, 40cpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WAS40 custom property using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/  
JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
```

Example output:

```
was4DS1(cells/mycell/nodes/mynodes:resources.xml$WAS40DataSource_1)
```

5.1+ Using Jython:

```
new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/  
JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')  
print new40ds
```

- Get required attributes:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newsds propertySet]
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#J2EEResourcePropertySet_9)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newds, 'propertySet')  
print propSet
```

- Get required attribute:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

Example output:

Attribute	Type
name	String

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up required attributes:

Using Jacl:

```
set name [list name RP5]  
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP5']  
rpAttrs = [name]
```

- Create J2EE Resource Property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

Example output:

```
RP5(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_9)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new JMS provider using wsadmin

- Identify the parent ID:

Using Jacl:

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

Example output:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

5.1+ Using Jython:

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

- Get required attributes:

Using Jacl:

```
$AdminConfig required JMSProvider
```

Example output:

Attribute	Type
name	String
externalInitialContextFactory	String
externalProviderURL	String

5.1+ Using Jython:

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

- 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]
```

Example output:

```
{name JMSP1} {externalInitialContextFactory  
{Put the external initial context factory here } }  
{externalProviderURL {Put the external provider URL here}}
```

5.1+ 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
```

- Create the JMS provider:

Using Jacl:

```
set newjmsp [$AdminConfig create JMSProvider $node $jmspAttrs]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.create('JMSProvider', node, jmspAttrs)
print newjmsp
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new JMS destination using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required GenericJMSDestination
```

Example output:

Attribute	Type
name	String
jndiName	String
externalJNDIName	String

5.1+ Using Jython:

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

- 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]
```

Example output:

```
{name JMSD1} {jndiName jms/JMSDestination1} {externalJNDIName jms/extJMSD1}
```

5.1+ Using Jython:

```
name = ['name', 'JMSD1']
jndi = ['jndiName', 'jms/JMSDestination1']
extJndi = ['externalJNDIName', 'jms/extJMSD1']
jmsdAttrs = [name, jndi, extJndi]
print jmsdAttrs
```

- Create generic JMS destination:

Using Jacl:

```
$AdminConfig create GenericJMSDestination $newjmsp $jmsdAttrs
```

Example output:

```
JMSD1(cells/mycell/nodes/mynode:resources.xml#GenericJMSDestination_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('GenericJMSDestination', newjmsp, jmsdAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1 + Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new JMS connection using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1 + Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required GenericJMSConnectionFactory
```

Example output:

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

5.1 + Using Jython:

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

- 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}
```

5.1 + Using Jython:

```
name = ['name', 'JMSCF1']
jndi = ['jndiName', 'jms/JMSConnFact1']
extJndi = ['externalJNDIName', 'jms/extJMSCF1']
jmscfAttrs = [name, jndi, extJndi]
print jmscfAttrs
```

5.1 + Example output:

```
[[name, JMSCF1], [jndiName, jms/JMSConnFact1], [externalJNDIName, jms/extJMSCF1]]
```

- Create generic JMS connection factory:

Using Jacl:

```
$AdminConfig create GenericJMSConnectionFactory $newjmsp $jmscfAttrs
```

Example output:

```
JMSCF1(cells/mycell/nodes/mynode:resources.xml#GenericJMSConnectionFactory_1)
```

5.1 + Using Jython:

```
print AdminConfig.create('GenericJMSConnectionFactory', newjmsp, jmscfAttrs)
```

- Save the changes with the following command:

Using Jacl:
\$AdminConfig save

5.1+ Using Jython:
AdminConfig.save()

Example: Configuring a new WebSphere queue connection factory using wsadmin:

- Identify the parent ID:

Using Jacl:
set newjmsp [\$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')  
print newjmsp
```

- Get required attributes:

Using Jacl:
\$AdminConfig required WASQueueConnectionFactory

Example output:

Attribute	Type
name	String
jndiName	String

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
name = ['name', 'WASQCF']  
jndi = ['jndiName', 'jms/WASQCF']  
mqcfAttrs = [name, jndi]  
print mqcfAttrs
```

5.1+ Example output:

```
[[name, WASQCF], [jndiName, jms/WASQCF]]
```

- Create was queue connection factories:

Using Jacl:
\$AdminConfig create WASQueueConnectionFactory \$newjmsp \$mqcfAttrs

Example output:

```
WASQCF(cells/mycell/nodes/mynode:resources.xml#WASQueueConnectionFactory_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('WASQueueConnectionFactory', newjmsp, mqcfAttrs)
```

- Save the changes with the following command:

Using Jacl:
\$AdminConfig save

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WebSphere topic connection factory using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')  
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required WASTopicConnectionFactory
```

Example output:

Attribute	Type
name	String
jndiName	String
port	ENUM(DIRECT, QUEUED)

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
name = ['name', 'WASTCF']  
jndi = ['jndiName', 'jms/WASTCF']  
port = ['port', 'QUEUED']  
mtcfAttrs = [name, jndi, port]  
print mtcfAttrs
```

5.1+ Example output:

```
[[name, WASTCF], [jndiName, jms/WASTCF], [port, QUEUED]]
```

- Create was topic connection factories:

Using Jacl:

```
$AdminConfig create WASTopicConnectionFactory $newjmsp $mtcfAttrs
```

Example output:

```
WASTCF(cells/mycell/nodes/mynode:resources.xml#WASTopicConnectionFactory_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('WASTopicConnectionFactory', newjmsp, mtcfAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```


5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WebSphere queue using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSPProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')  
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required WASQueue
```

Example output:

Attribute	Type
name	String
jndiName	String

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
name = ['name', 'WASQ1']  
jndi = ['jndiName', 'jms/WASQ1']  
wqAttrs = [name, jndi]  
print wqAttrs
```

5.1+ Example output:

```
[[name, WASQ1], [jndiName, jms/WASQ1]]
```

- Create was queue:

Using Jacl:

```
$AdminConfig create WASQueue $newjmsp $wqAttrs
```

Example output:

```
WASQ1(cells/mycell/nodes/mynode:resources.xml#WASQueue_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('WASQueue', newjmsp, wqAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new WebSphere topic using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSP1_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required WASTopic
```

Example output:

Attribute	Type
name	String
jndiName	String
topic	String

5.1+ Using Jython:

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

- 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}}
```

5.1+ Using Jython:

```
name = ['name', 'WAST1']
jndi = ['jndiName', 'jms/WAST1']
topic = ['topic', "Put your topic here"]
wtAttrs = [name, jndi, topic]
print wtAttrs
```

5.1+ Example output:

```
[[name, WAST1], [jndiName, jms/WAST1], [topic, "Put your topic here"]]
```

- Create was topic:

Using Jacl:

```
$AdminConfig create WASTopic $newjmsp $wtAttrs
```

Example output:

```
WAST1(cells/mycell/nodes/mynode:resources.xml#WASTopic_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('WASTopic', newjmsp, wtAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new MQ queue connection factory using wsadmin:

- Identify the parent ID:
Using Jacl:
set newjmsp [\$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
Example output:
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required MQQueueConnectionFactory
```

Example output:

Attribute name	Type
jndiName	String
	String

5.1+ Using Jython:

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

- Set up required attributes:

Using Jacl:

```
set name [list name MQQCF]
set jndi [list jndiName jms/MQQCF]
set mqccfAttrs [list $name $jndi]
```

Example output:

```
{name MQQCF} {jndiName jms/MQQCF}
```

5.1+ Using Jython:

```
name = ['name', 'MQQCF']
jndi = ['jndiName', 'jms/MQQCF']
mqccfAttrs = [name, jndi]
print mqccfAttrs
```

5.1+ Example output:

```
[[name, MQQCF], [jndiName, jms/MQQCF]]
```

- Create MQ queue connection factory:

Using Jacl:

```
$AdminConfig create MQQueueConnectionFactory $newjmsp $mqccfAttrs
```

Example output:

```
MQQCF(cells/mycell/nodes/mynode:resources.xml#MQQueueConnectionFactory_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('MQQueueConnectionFactory', newjmsp, mqccfAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new MQ topic connection factory using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required MQTopicConnectionFactory
```

Example output:

Attribute	Type
name	String
jndiName	String

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
name = ['name', 'MQTCF']
jndi = ['jndiName', 'jms/MQTCF']
mqtcfAttrs = [name, jndi]
print mqtcfAttrs
```

5.1+ Example output:

```
[[name, MQTCF], [jndiName, jms/MQTCF]]
```

- Create mq topic connection factory:

Using Jacl:

```
$AdminConfig create MQTopicConnectionFactory $newjmsp $mqtcfAttrs
```

Example output:

```
MQTCF(cells/mycell/nodes/mynode:resources.xml#MQTopicConnectionFactory_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('MQTopicConnectionFactory', newjmsp, mqtcfAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new MQ queue using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required MQQueue
```

Example output:

```
Attribute      Type
name           String
jndiName       String
baseQueueName  String
```

5.1+ Using Jython:

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

- 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}}
```

5.1+ Using Jython:

```
name = ['name', 'MQQ']
jndi = ['jndiName', 'jms/MQQ']
baseQN = ['baseQueueName', "Put the base queue name here"]
mqqAttrs = [name, jndi, baseQN]
print mqqAttrs
```

5.1+ Example output:

```
[[name, MQQ], [jndiName, jms/MQQ], [baseQueueName, "Put the base queue name here"]]
```

- Create MQ queue factory:

Using Jacl:

```
$AdminConfig create MQQueue $newjmsp $mqqAttrs
```

Example output:

```
MQQ(cells/mycell/nodes/mynode:resources.xml#MQQueue_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('MQQueue', newjmsp, mqqAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new MQ topic using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

5.1+ Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required MQTopic
```

Example output:

Attribute	Type
name	String
jndiName	String
baseTopicName	String

5.1+ Using Jython:

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

- 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}}
```

5.1+ Using Jython:

```
name = ['name', 'MQT']
jndi = ['jndiName', 'jms/MQT']
baseTN = ['baseTopicName', "Put the base topic name here"]
mqtAttrs = [name, jndi, baseTN]
print mqtAttrs
```

5.1+ Example output:

```
[[name, MQT], [jndiName, jms/MQT], [baseTopicName, "Put the base topic name here"]]
```

- Create MQ topic factory:

Using Jacl:

```
$AdminConfig create MQTopic $newjmsp $mqtAttrs
```

Example output:

```
MQT(cells/mycell/nodes/mynode:resources.xml#MQTopic_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('MQTopic', newjmsp, mqtAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new mail provider using wsadmin

- Identify the parent ID:

Using Jacl:

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

Example output:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

5.1+ Using Jython:

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

- Get required attributes:

Using Jacl:

```
$AdminConfig required MailProvider
```

Example output:

Attribute	Type
name	String

5.1+ Using Jython:

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

- Set up required attributes:

Using Jacl:

```
set name [list name MP1]
set mpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'MP1']
mpAttrs = [name]
```

- Create the mail provider:

Using Jacl:

```
set newmp [$AdminConfig create MailProvider $node $mpAttrs]
```

Example output:

```
MP1(cells/mycell/nodes/mynode:resources.xml#MailProvider_1)
```

5.1+ Using Jython:

```
newmp = AdminConfig.create('MailProvider', node, mpAttrs)
print newmp
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new mail session using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
```

Example output:

```
MP1(cells/mycell/nodes/mynode:resources.xml#MailProvider_1)
```

5.1+ Using Jython:

```
newmp = AdminConfig.create('MailProvider', node, mpAttrs)
print newmp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required MailSession
```

Example output:

Attribute	Type
name	String
jndiName	String

5.1+ Using Jython:

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

- 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}
```

5.1+ Using Jython:

```
name = ['name', 'MS1']
jndi = ['jndiName', 'mail/MS1']
msAttrs = [name, jndi]
print msAttrs
```

5.1+ Example output:

```
[[name, MS1], [jndiName, mail/MS1]]
```

- Create the mail session:

Using Jacl:

```
$AdminConfig create MailSession $newmp $msAttrs
```

Example output:

```
MS1(cells/mycell/nodes/mynode:resources.xml#MailSession_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('MailSession', newmp, msAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new protocol provider using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
```

Example output:

```
MP1(cells/mycell/nodes/mynode:resources.xml#MailProvider_1)
```

5.1+ Using Jython:

```
newmp = AdminConfig.create('MailProvider', node, mpAttrs)
print newmp
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required ProtocolProvider
```

Example output:

Attribute	Type
protocol	String
classname	String

5.1+ Using Jython:

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


- 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}
```

5.1+ Using Jython:

```
protocol = ['protocol', "Put the protocol here"]
classname = ['classname', "Put the class name here"]
ppAttrs = [protocol, classname]
print ppAttrs
```

5.1+ Example output:

```
[[protocol, protocol1], [classname, classname1]]
```

- Create the protocol provider:

Using Jacl:

```
$AdminConfig create ProtocolProvider $newmp $ppAttrs
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#ProtocolProvider_4)
```

5.1+ Using Jython:

```
print AdminConfig.create('ProtocolProvider', newmp, ppAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new custom property using wsadmin:

- Identify the parent ID:

Using Jacl:

```
set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
```

Example output:

```
MP1(cells/mycell/nodes/mynode:resources.xml#MailProvider_1)
```

5.1+ Using Jython:

```
newmp = AdminConfig.create('MailProvider', node, mpAttrs)
print newmp
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newmp propertySet]
```

Example output:

```
(cells/mycell/nodes/mynode:resources.xml#PropertySet_2)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newmp, 'propertySet')
print propSet
```

- Get required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

Example output:

```
Attribute      Type
name          String
```

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name CP1]
set cpAttrs [list $name]
```

Example output:

```
{name CP1}
```

5.1+ Using Jython:

```
name = ['name', 'CP1']
cpAttrs = [name]
print cpAttrs
```

5.1+ Example output:

```
[[name, CP1]]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $cpAttrs
```

Example output:

```
CP1(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_2)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, cpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new resource environment provider using wsadmin

An example configuring a new resource environment provider follows:

- Identify the parent ID and assign it to the node variable.

Using Jacl:

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

An example of this output follows:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

5.1+ Using Jython:

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

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required ResourceEnvironmentProvider
```

An example of this output follows:

```
Attribute      Type
name          String
```

5.1+ Using Jython:

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

- Set up the required attributes and assign it to the repAttrs variable:

Using Jacl:

```
set n1 [list name REP1]
set repAttrs [list $name]
```

5.1+ Using Jython:

```
n1 = ['name', 'REP1']
repAttrs = [n1]
```

- Create a new resource environment provider:

Using Jacl:

```
set newrep [$AdminConfig create ResourceEnvironmentProvider $node $repAttrs]
```

An example of this output follows:

```
REP1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvironmentProvider_1)
```

5.1+ Using Jython:

```
newrep = AdminConfig.create('ResourceEnvironmentProvider', node, repAttrs)
print newrep
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for a resource environment provider

using wsadmin: An example configuring a new custom property for a resource environment provider follows:

- Identify the parent ID and assign it to the newrep variable.

Using Jacl:

```
set newrep [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
```

An example of this output follows:

```
REP1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvironmentProvider_1)
```

5.1+ Using Jython:

```
newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
print newrep
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

```
Attribute      Type
name          String
```

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes and assign it to the repAttrs variable:

Using Jacl:

```
set name [list name RP]
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP']
rpAttrs = [name]
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newrep propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#PropertySet_1)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newrep, 'propertySet')
print propSet
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RP(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new referenceable using wsadmin: An example configuring a new referenceable follows:

- Identify the parent ID and assign it to the newrep variable.

Using Jacl:

```
set newrep [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
```

An example of this output follows:

```
REP1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvironmentProvider_1)
```

5.1+ Using Jython:

```
newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
print newrep
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required Referenceable
```

An example of this output follows:

```
Attribute      Type
factoryClassname String
classname      String
```

5.1+ Using Jython:

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

- Set up the required attributes:

Using Jacl:

```
set fcn [list factoryClassname REP1]
set cn [list classname NM1]
set refAttrs [list $fcn $cn]
```

An example of this output follows:

```
{factoryClassname {REP1}} {classname {NM1}}
```

5.1+ Using Jython:

```
fcn = ['factoryClassname', 'REP1']
cn = ['classname', 'NM1']
refAttrs = [fcn, cn]
print refAttrs
```

- Create a new referenceable:

Using Jacl:

```
set newref [$AdminConfig create Referenceable $newrep $refAttrs]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#Referenceable_1)
```

5.1+ Using Jython:

```
newref = AdminConfig.create('Referenceable', newrep, refAttrs)
print newref
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new resource environment entry using wsadmin: An example configuring a new resource environment entry follows:

- Identify the parent ID and assign it to the newrep variable.

Using Jacl:

```
set newrep [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
```

An example of this output follows:

```
REP1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvironmentProvider_1)
```

5.1+ Using Jython:

```
newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
print newrep
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required ResourceEnvEntry
```

An example of this output follows:

```
Attribute      Type
name           String
jndiName       String
referenceable   Referenceable@
```

5.1+ Using Jython:

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

- 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]
```

5.1+ Using Jython:

```

name = ['name', 'REE1']
jndiName = ['jndiName', 'myjndi']
newref = AdminConfig.getid('/Cell:mycell/Node:mynode/Referenceable:/')
ref = ['referenceable', newref]
reeAttrs = [name, jndiName, ref]

```

- Create the resource environment entry:

Using Jacl:

```
$AdminConfig create ResourceEnvEntry $newrep $reeAttrs
```

An example of this output follows:

```
REE1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvEntry_1)
```

5.1 + Using Jython:

```
print AdminConfig.create('ResourceEnvEntry', newrep, reeAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1 + Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for resource environment entries using wsadmin:

An example configuring a new custom property for a resource environment entry follows:

- Identify the parent ID and assign it to the newree variable.

Using Jacl:

```
set newree [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/]
```

An example of this output follows:

```
REE1(cells/mycell/nodes/mynode:resources.xml#ResourceEnvEntry_1)
```

5.1 + Using Jython:

```
newree = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/')
print newree
```

- Create the J2EE custom property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newree propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#J2EEResourcePropertySet_5)
```

5.1 + Using Jython:

```
propSet = AdminConfig.showAttribute(newree, 'propertySet')
print propSet
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

```
Attribute      Type
name           String
```

5.1 + Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name RP1]
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP1']
rpAttrs = [name]
```

- Create the J2EE custom property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RPI(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new URL provider using wsadmin

An example configuring a new URL provider follows:

- Identify the parent ID and assign it to the node variable.

Using Jacl:

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

An example of this output follows:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

5.1+ Using Jython:

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

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required URLProvider
```

An example of this output follows:

Attribute	Type
streamHandlerClassName	String
protocol	String
name	String

5.1+ Using Jython:

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

- 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]
```

An example of this output follows:

```
{name URLP1} {streamHandlerClassName
{Put the stream handler classname here}}
{protocol {Put the protocol here}}
```

5.1+ 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

```

5.1 + An example of this output follows:

```

[[name, URLP1], [streamHandlerClassName,
"Put the stream handler classname here"],
[protocol, "Put the protocol here"]]

```

- Create a URL provider:

Using Jacl:

```
$AdminConfig create URLProvider $node $urlpAttrs
```

An example of this output follows:

```
URLP1(cells/mycell/nodes/mynode:resources.xml#URLProvider_1)
```

5.1 + Using Jython:

```
print AdminConfig.create('URLProvider', node, urlpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1 + Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for URL providers using wsadmin: An example configuring a new custom property for URL providers follows:

- Identify the parent ID and assign it to the newurlp variable.

Using Jacl:

```
set newurlp [$AdminConfig getid /Cell:mycell/Node:mynode/URLProvider:URLP1/]
```

An example of this output follows:

```
URLP1(cells/mycell/nodes/mynode:resources.xml#URLProvider_1)
```

5.1 + Using Jython:

```
newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
print newurlp
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newurlp propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#PropertySet_7)
```

5.1 + Using Jython:

```
propSet = AdminConfig.showAttribute(newurlp, 'propertySet')
print propSet
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

```
Attribute      Type
name           String
```

5.1 + Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```


- Set up the required attributes:

Using Jacl:

```
set name [list name RP2]
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP2']
rpAttrs = [name]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RP2(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new URL using wsadmin: An example configuring a new URL follows:

- Identify the parent ID and assign it to the newurlp variable.

Using Jacl:

```
set newurlp [$AdminConfig getid /Cell:mycell/Node:mynode/URLProvider:URLP1/]
```

An example of this output follows:

```
URLP1(cells/mycell/nodes/mynode:resources.xml#URLProvider_1)
```

5.1+ Using Jython:

```
newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
print newurlp
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required URL
```

An example of this output follows:

Attribute	Type
name	String
spec	String

5.1+ Using Jython:

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

- 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]
```

An example of this output follows:

```
{name URL1} {spec {Put the spec here}}
```

5.1+ Using Jython:

```
name = ['name', 'URL1']
spec = ['spec', "Put the spec here"]
urlAttrs = [name, spec]
```

5.1+ An example of this output follows:

```
[[name, URL1], [spec, "Put the spec here"]]
```

- Create a URL:

Using Jacl:

```
$AdminConfig create URL $newurlp $urlAttrs
```

An example of this output follows:

```
URL1(cells/mycell/nodes/mynode:resources.xml#URL_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('URL', newurlp, urlAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for URLs using wsadmin: An example configuring a new custom property for a URL follows:

- 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/]
```

An example of this output follows:

```
URL1(cells/mycell/nodes/mynode:resources.xml#URL_1)
```

5.1+ Using Jython:

```
newurl = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/URL:URL1/')
print newurl
```

- Create a J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newurl propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#J2EEResourcePropertySet_7)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newurl, 'propertySet')
print propSet
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

Attribute	Type
name	String

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name RP3]
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP3']
rpAttrs = [name]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RP3(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_7)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new J2C resource adapter using wsadmin

An example configuring a new J2C resource adapter follows:

- Identify the parent ID and assign it to the node variable.

Using Jacl:

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

An example of this output follows:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

5.1+ Using Jython:

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

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2CResourceAdapter
```

An example of this output follows:

Attribute	Type
name	String

5.1+ Using Jython:

```
print AdminConfig.required('J2CResourceAdapter')
```

- Set up the required attributes:

Using Jacl:

```
set rarFile /currentScript/cicseci.rar
set option [list -rar.name RAR1]
```

5.1+ Using Jython:

```
rarFile = '/currentScript/cicseci.rar'
option = '[-rar.name RAR1]'
```

- Create a resource adapter:

Using Jacl:

```
set newra [$AdminConfig installResourceAdapter $rarFile mynode $option]
```

An example of this output follows:

```
RAR1(cells/mycell/nodes/mynode:resources.xml#J2CResourceAdapter_1)
```

5.1+ Using Jython:

```
newra = AdminConfig.installResourceAdapter(rarFile, 'mynode', option)
print newra
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for J2C resource adapters using

wsadmin: An example configuring a new custom property for a J2C resource adapters follows:

- Identify the parent ID and assign it to the newra variable.

Using Jacl:

```
set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
```

An example of this output follows:

```
RAR1(cells/mycell/nodes/mynode:resources.xml#J2CResourceAdapter_1)
```

5.1+ Using Jython:

```
newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
print newra
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newra propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#PropertySet_8)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newra, 'propertySet')
print propSet
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

Attribute	Type
name	String

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name RP4]
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP4']
rpAttrs = [name]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RP4(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_8)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring a new J2C connection factory using wsadmin: An example configuring a new J2C resource factory follows:

- Identify the parent ID and assign it to the newra variable.

Using Jacl:

```
set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
```

An example of this output follows:

```
RAR1(cells/mycell/nodes/mynode:resources.xml#J2CResourceAdapter_1)
```

5.1+ Using Jython:

```
newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')  
print newra
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2CConnectionFactory
```

An example of this output follows:

Attribute name	Type
	String

5.1+ Using Jython:

```
print AdminConfig.required('J2CConnectionFactory')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name J2CCF1]  
set j2ccfAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'J2CCF1']  
j2ccfAttrs = [name]
```

- Create a connection factory:

Using Jacl:

```
$AdminConfig create J2CConnectionFactory $newra $j2ccfAttrs
```

An example of this output follows:

```
J2CCF1(cells/mycell/nodes/mynode:resources.xml#J2CConnectionFactory_1)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2CConnectionFactory', newra, j2ccfAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring custom properties for J2C connection factories using wsadmin: An example configuring a new custom property for a J2C resource factories follows:

- Identify the parent ID and assign it to the newcf variable:

Using Jacl:

```
set newcf [$AdminConfig getid /J2CConnectionFactory:J2CCF1/]
```

An example of this output follows:

```
J2CCF1(cells/mycell/nodes/mynode:resources.xml#J2CConnectionFactory_1)
```

5.1+ Using Jython:

```
newcf = AdminConfig.getid('/J2CConnectionFactory:J2CCF1/')  
print newcf
```

- Get the J2EE resource property set:

Using Jacl:

```
set propSet [$AdminConfig showAttribute $newcf propertySet]
```

An example of this output follows:

```
(cells/mycell/nodes/mynode:resources.xml#PropertySet_8)
```

5.1+ Using Jython:

```
propSet = AdminConfig.showAttribute(newcf, 'propertySet')  
print propSet
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

An example of this output follows:

Attribute	Type
name	String

5.1+ Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

- Set up the required attributes:

Using Jacl:

```
set name [list name RP4]  
set rpAttrs [list $name]
```

5.1+ Using Jython:

```
name = ['name', 'RP4']  
rpAttrs = [name]
```

- Create a J2EE resource property:

Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

An example of this output follows:

```
RP4(cells/mycell/nodes/mynode:resources.xml#J2EEResourceProperty_8)
```

5.1+ Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring new J2C authentication data entries using wsadmin: An example configuring new J2C authentication data entries follows:

- Identify the parent ID and assign it to the security variable.

Using Jacl:

```
set security [$AdminConfig getid /Security:mysecurity/]
```

5.1+ Using Jython:

```
security = AdminConfig.getid('/Security:mysecurity/')
```

- Identify the required attributes:

Using Jacl:

```
$AdminConfig required JAASAuthData
```

An example of this output follows:

Attribute	Type
alias	String
userId	String
password	String

5.1+ Using Jython:

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

- 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]
```

An example of this output follows:

```
{alias myAlias} {userId myid} {password secret}
```

5.1+ Using Jython:

```
alias = ['alias', 'myAlias']
userid = ['userId', 'myid']
password = ['password', 'secret']
jaasAttrs = [alias, userid, password]
```

5.1+ An example of this output follows:

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

- Create JAAS authentication data:

Using Jacl:

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

An example of this output follows:

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

5.1+ Using Jython:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Enabling and disabling global security with a profile

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
```

An example of this output follows:

Syntax: securityon user password

5.1+ Using Jython:

```
securityon()
```

5.1+ An example of this output follows:

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
```

5.1+ Using Jython:

```
securityon('user1', 'password')
```

- To disable global security based on the LocalOS registry, use the following procedure call:

Using Jacl:

```
securityoff
```

5.1+ Using Jython:

```
securityoff()
```

Example: Enabling and disabling Java 2 security using wsadmin

An example of enabling and disabling Java 2 security follows:

- Identify the security configuration object and assign it to the security variable:

Using Jacl:

```
set security [$AdminConfig list Security]
```

An example of this output follows:

```
(cells/mycell:security.xml#Security_1)
```

5.1+ Using Jython:

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

- Modify the enforceJava2Security attribute.

To enable Java 2 security:

Using Jacl:

```
$AdminConfig modify $security {{enforceJava2Security true}}
```

5.1+ Using Jython:

```
AdminConfig.modify(security, [['enforceJava2Security', 'true']])
```

To disable Java 2 security:

Using Jacl:

```
$AdminConfig modify $security {{enforceJava2Security false}}
```

5.1+ Using Jython:

```
AdminConfig.modify(security, [['enforceJava2Security', 'false']])
```

- Save the changes with the following command:

Using Jacl:


```
$AdminConfig save
```

5.1 + Using Jython:

```
AdminConfig.save()
```

Managing running objects with scripting

Operation management scripts use the AdminControl object to communicate with the MBeans that represent running objects. You can use the AdminControl object to list running objects and their attributes, invoke actions on running objects, obtain help, and obtain dynamic information about running MBeans.

1. Decide how you want to execute the script. If you want to run the script immediately from the command line, enter it surrounded by quotes as a parameter to the **wsadmin -c** command. To save the script for repeated use, compose it in a file and execute it with the **wsadmin -f** command. If you want to compose and run the script interactively, issue the **wsadmin** command without the **-c** or **-f** flags. For more information about executing scripts, see *Launching scripting clients*
2. Write an AdminControl script command statement to perform a management task, for example:

Using Jacl:

```
$AdminControl command
```

5.1 + Using Jython:

```
AdminControl.command()
```

Specifying running objects using the wsadmin tool

1. Invoke the AdminControl object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. 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]
```

5.1 + 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 returns 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,server:server1*]
```

5.1+ Using Jython:

```
server1 = AdminControl.completeObjectName('node=mynode,server:server1*')
```

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

Using Jacl:

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

5.1+ 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]*

3. 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 **index** 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]
```

5.1+ 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

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

1. Invoke the AdminControl object commands interactively, in a script, or use the **wsadmin -c** tool from an operating system command prompt.
2. Specify a running object.
3. Use the **attributes** command to display the attributes of the running object:

Using Jacl:

```
$Help attributes MBeanObjectName
```

5.1+ 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

4. Use the **operations** command to find out the operations supported by the MBean:

Using Jacl:

```
$Help operations MBeanObjectname
```

or

```
$Help operations MBeanObjectname operationName
```

5.1+ 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
```

5.1+ Using Jython:

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

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

Using Jacl:

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

5.1+ Using Jython:

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

Performing operations on running objects using the wsadmin tool

1. Invoke the AdminControl object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Obtain the object name of the running object with the following command:

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

<i>name</i>	is a fragment of the object name. It is used to find the matching object name. For example: type=Server,name=server1,* . It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, etc.
-------------	---

3. Issue the following command:

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

4. Invoke the operation with the following command:

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

1. Invoke the AdminControl object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Obtain the name of the running object with the following command:

Using Jacl:

```
$AdminControl completeObjectName name
```

5.1+ 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
<i>name</i>	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.

3. Issue the following command:

Using Jacl:

```
set ts1 [${AdminControl completeObjectName name}
```

5.1+ 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

<i>name</i>	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.
-------------	--

4. Modify the running object with the following command:

Using Jacl:

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

5.1 + 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}


```

5.1 + Using Jython list:

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

5.1 + 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.

Related reference

“Example: Identifying running objects” on page 201

Operation management examples with wsadmin

There are examples that illustrate how to manage running objects using wsadmin. Use these examples to identify running objects, modify them and invoke actions on them with the AdminControl object. Basic knowledge of the syntax for the Jacl scripting language is helpful in order to understand and modify the examples.

5.1+ You can also download a set of sample scripts from IBM developerWorks. To download the sample scripts, see Sample scripts for WebSphere Application Server (<http://www-106.ibm.com/developerworks/websphere/library/samples/SampleScripts.html>).

Example: Representing lists and javax.management.AttributeList objects with strings when scripting

Represent lists and javax.management.AttributeList objects using strings when scripting. These objects have different syntax conventions depending on the scripting language and whether the list or AttributeList object is input or output. The Java Management Extensions (JMX) Specification states that:

- The Attribute class represents a single attribute-value pair
- The AttributeList class represents a list of attribute-value pairs

The Attribute and AttributeList objects are typically used for conveying the attribute values of an MBean, as the result of a getter operation, or as the argument of a setter operation.

Jacl

When using the Jacl interface, construct lists using the standard Jacl list syntax. For example, a list of two attribute names might look like: {passivationDirectory inactivePoolCleanupInterval} On output, the wsadmin tool does not display the enclosing brackets, and you can manipulate the result as a Jacl list. An AttributeList object is represented by a Jacl list. The outer enclosing list represents the entire AttributeList class and can have any number of interior lists. The interior lists each have a length of two, where the first element is the attribute name, and the second element is a string representation of the attribute value. For example, {{passivationDirectory /temp}} is an attribute list containing a single attribute name and value, and {{traceSpecification com.ibm.*=all=enabled} {ringBufferSize 0}} contains two attributes. Enter input (in a script) string AttributeLists exactly like this. If the value of the attribute contains a space, then enclose the value in another set of curly braces, or in double quotes: {{passivationDirectory "/My Folder/temp"}} On output, the outer set of parenthesis does not display, which makes the output value a Jacl list that the script can easily manipulate.

Jython 5.1+

When using the Jython interface, construct lists using the standard Jython list syntax. For example, a list of two attribute names might look like the following: [['passivationDirectory', 'inactivePoolCleanupInterval']]. You can manipulate the result as a Jython list. An AttributeList object is represented by a Jython list. The outer enclosing list represents the entire AttributeList class and can have any number of interior lists. The interior lists each have a length of two where the first element is the attribute name, and the second element is a string representation of the attribute value. For example, [['passivationDirectory', '/temp']] is an attribute list containing a single attribute name and value, and [['traceSpecification', 'com.ibm.*=all=enabled'], ['ringBufferSize', 0]] contains two attributes. Enter input (in a script) string AttributeLists exactly like this. If the value of the attribute contains a space, then enclose the value in double quote: [['passivationDirectory', "/My Folder/temp"]].

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 *
```

5.1+ 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:*
```

5.1+ 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,*
```

5.1+ 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
```

5.1+ 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,*
```

5.1+ Using Jython:

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

Example: Dumping threads in a server process

Use the AdminControl object to dump the Java threads of a running server.

- For example:

Using Jacl:

```
set jvm [$AdminControl completeObjectName type=JVM,process=server1,*]
$AdminControl invoke $jvm dumpThreads
```

5.1+ Using Jython:

```
jvm = AdminControl.completeObjectName('type=JVM,process=server1,*')
AdminControl.invoke(jvm, 'dumpThreads')
```

This example produces a Java core file. You can use this file for problem determination.

Example: Setting up profiles to make tracing easier when scripting

Set up a profile to make tracing easier. The following profile example turns tracing on and off:

Using Jacl:

```
proc ton {} {
  global AdminControl
  set ts [lindex [$AdminControl queryNames type=TraceService,*] 0]
  $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=enabled
}

proc toff {} {
  global AdminControl
  set ts [lindex [$AdminControl queryNames type=TraceService,*] 0]
  $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=disabled
}

proc dt {} {
```

```

global AdminControl
set jvm [lindex [$AdminControl queryNames type=JVM,*] 0]
$AdminControl invoke $jvm dumpThreads
}

```

5.1+ Using Jython:

```

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

def ton():
    global lineSeparator
    ts = AdminControl.queryNames('type=TraceService,*').split(lineSeparator)[0]

    AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.=all=enabled')

def toff():
    global lineSeparator
    ts = AdminControl.queryNames('type=TraceService,*').split(lineSeparator)[0]

    AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=disabled')

def dt():
    global lineSeparator
    jvm = AdminControl.queryNames('type=JVM,*').split(lineSeparator)[0]
    AdminControl.invoke(jvm, 'dumpThreads')

```

If you start the wsadmin tool with this profile, 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, see the [Launching Scripting Clients](#) article.

Example: Starting a server using wsadmin

The following example starts an application server with the node specified.

- The following command starts server1 in mynode node:

Using Jacl:

```
$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"
```

5.1+ Using Jython:

```
print AdminControl.startServer('server1', 'mynode')
```

- The **startServer** command has several command syntax options. If you have Network Deployment installation, you have to use one of following:

Using Jacl:

```
$AdminControl startServer serverName nodeName
```

```
$AdminControl startServer serverName nodeName waitTime
```

5.1+ Using Jython:

```
AdminControl.startServer('serverName', 'nodeName')
```

```
AdminControl.startServer('serverName', 'nodeName', waitTime)
```

- If you have an application server base installation, you can use the following syntax in addition to the previous syntax:

Using Jacl:

```
$AdminControl startServer serverName  
$AdminControl startServer serverName waitTime
```

5.1+ Using Jython:

```
AdminControl.startServer('serverName')  
AdminControl.startServer('serverName', waitTime)
```

Example: Stopping a server using wsadmin

The following example stops an application server with the node specified.

- The following command stops *server1* in node *mynode*.

Using Jacl:

```
$AdminControl stopServer server1 mynode
```

Example output:

```
WASX7337I: Invoked stop for server "server1" Waiting for stop completion.  
WASX7264I: Stop completed for server "server1" on node "mynode"
```

5.1+ Using Jython:

```
print AdminControl.stopServer('server1', 'mynode')
```

- The stop command has several command syntaxes.

If you have Network Deployment installation, use the one of following command syntax:

Using Jacl:

```
$AdminControl stopServer serverName nodeName
```

```
$AdminControl stopServer serverName nodeName immediate
```

5.1+ Using Jython:

```
AdminControl.stopServer('serverName', 'nodeName')
```

```
AdminControl.stopServer('serverName', 'nodeName', 'immediate')
```

If you have application server base installation, you can use the following syntax, in addition to the previous syntax:

Using Jacl:

```
$AdminControl stopServer serverName
```

```
$AdminControl stopServer serverName immediate
```

5.1+ Using Jython:

```
AdminControl.stopServer('serverName')
```

```
AdminControl.stopServer('serverName', 'immediate')
```

Example: Querying the server state using the wsadmin tool

The following example queries the server state.

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

Using Jacl:

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

5.1+ Using Jython:

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

This command returns the server MBean that matches the partial object name string.

Example output:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#
Server_1,type=Server,node=mynode,process=server1,
processType=ManagedProcess
```

- Query for the state attribute.

Using Jacl:

```
$AdminControl getAttribute $server state
```

5.1+ Using Jython:

```
print AdminControl.getAttribute(server, 'state')
```

The **getAttribute** command returns the value of a single attribute.

Example output:

```
STARTED
```

Example: Querying the product identification using wsadmin

The following example queries the product version information.

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

Using Jacl:

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

Example output:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#
Server_1,type=Server,node=mynode,process=server1,
processType=ManagedProcess
```

5.1+ Using Jython:

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

- Query the server version. The product 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 Jacl:

```
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

Example: Starting a listener port using wsadmin

The following example starts a listener port on an application server.

- 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,*]
```

5.1+ Using Jython:

```
lPorts = AdminControl.queryNames('type=ListenerPort,cell=  
mycell,node=mynode,process=server1,*')  
print lPorts
```

This command returns a list of listener port MBeans.

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
```

- Start the listener port if it is not started with the following example:

Using Jacl:

```
foreach lPort $lPorts {  
    set state [$AdminControl getAttribute $lport started]  
    if {$state == "false"} {  
        $AdminControl invoke $lPort start  
    }  
}
```

5.1+ 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')
```

5.1+ 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.

Example: Testing data source connection using wsadmin to call a method on the MBean

The following example tests a `dataSource`, to ensure a connection to the database.

- Identify the `DataSourceCfgHelper` MBean and assign it to the `dshelper` variable.

Using Jacl:

```
set dshelper [$AdminControl queryNames
type=DataSourceCfgHelper,process=server1,*]
```

Example output:

```
WebSphere:cell=mycell,name=DataSourceCfgHelper,
mbeanIdentifier=DataSourceCfgHelper,
type=DataSourceCfgHelper,node=mynode,process=server1
```

5.1+ Using Jython:

```
dshelper = AdminControl.queryNames('type=DataSourceCfgHelper,
process=server1*')
print dshelper
```

- Test the connection.

Using Jacl:

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

5.1+ Using Jython:

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

This example command invokes the `testConnectionToDataSource` operation on the MBean, passing in the classname, userid, password, database name, JDBC driver class path, language, and country.

Example output:

```
DSRA8025I: Successfully connected to DataSource
```

The Preferred Method

Instead of using the method `testConnectionToDataSource`, use the method `testConnection` and pass in the configuration ID of the data source. The `testConnection` method is not available in V5.0. The data source to be tested has to be saved in the master configuration.

```
set myds [$AdminConfig getid "/JDBCProvider:Sybase
12.0 JDBC Driver/DataSource:sybaseds/"]
$AdminControl invoke $dshelper testConnection $myds
```

To aid in making a batch program, the command returns a value instead of a message. A return value of **0** means success. A return value of **1 - n** means success, but with a number of warnings. If the process fails, you receive an exception message.

Example: Configuring transaction properties for a server using wsadmin

The following example configures the run-time transaction properties for an application server.

- Identify the transaction service MBean for the application server.

Using Jacl:

```
set ts [$AdminControl completeObjectName cell=mycell,node=
mynode,process=server1,type=TransactionService,*]
```

5.1+ Using Jython:

```
ts = AdminControl.completeObjectName('cell=mycell,node=mynode,
process=server1,type=TransactionService,*')
print ts
```

This command returns the transaction service MBean for server1.

Example output:

```
WebSphere:cell=mycell,name=TransactionService,mbeanIdentifier=
TransactionService,type=TransactionService,node=mynode,process=server1
```

- Modify the attributes. The following example is for the Windows operating system:

Using Jacl:

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

5.1+ Using Jython:

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

The clientInactivityTimeout is in seconds. The totalTranLifetimeTimeout is in milliseconds. A value of 0 in either attribute means no timeout limit.

Example: Starting a cluster using wsadmin

The following example starts a cluster:

- Identify the ClusterMgr MBean and assign it to the clusterMgr variable.

Using Jacl:

```
set clusterMgr [$AdminControl completeObjectName
cell=mycell,type=ClusterMgr,*]
```

5.1+ 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
```

- Refresh the list of clusters.

Using Jacl:

```
$AdminControl invoke $clusterMgr retrieveClusters
```

5.1+ Using Jython:

```
AdminControl.invoke(clusterMgr, 'retrieveClusters')
```

This command calls the retrieveClusters operation on the ClusterMgr MBean.

- Identify the Cluster MBean and assign it to the cluster variable.

Using Jacl:

```
set cluster [$AdminControl completeObjectName
cell=mycell,type=Cluster,name=cluster1,*]
```


5.1+ 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
```

- Start the cluster.

Using Jacl:

```
$AdminControl invoke $cluster start
```

5.1+ Using Jython:

```
AdminControl.invoke(cluster, 'start')
```

This command invokes the start operation on the Cluster MBean.

Example: Stopping a cluster using wsadmin

The following example stops a cluster:

- Identify the Cluster MBean and assign it to the cluster variable.

Using Jacl:

```
set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*
```

5.1+ 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
```

- Stop the cluster.

Using Jacl:

```
$AdminControl invoke $cluster stop
```

5.1+ Using Jython:

```
AdminControl.invoke(cluster, 'stop')
```

This command invokes the stop operation on the Cluster MBean.

Example: Querying cluster state using wsadmin

The following example queries the state of the cluster:

- Identify the Cluster MBean and assign it to the cluster variable.

Using Jacl:

```
set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*
```

5.1+ 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
```

- Query the cluster state.

Using Jacl:

```
$AdminControl getAttribute $cluster state
```

5.1+ Using Jython:

```
AdminControl.getAttribute(cluster, 'state')
```

This command returns the value of the run-time state attribute.

Example: Listing running applications on running servers using wsadmin

The following example lists all the running applications on all the running servers on each node of each cell.

- 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 $nodes {
12        #-----
13        # lines 16-20 find all the running
14        servers belonging to the cell
15        # and node, and process them one at a time
16        #-----
17        set cname [$AdminConfig showAttribute $cell name]
18        set nname [$AdminConfig showAttribute $node name]
19        set servs [$AdminControl queryNames
20        type=Server,cell=$cname,node=$nname,*]
21        puts "Number of running servers on node
22        $nname: [llength $servs]"
23        foreach server $servs {
24            #-----
25            # lines 25-31 get some attributes
26            from the server to display;
27            # invoke an operation on the server
28            JVM to display a property.
29            #-----
30            set sname [$AdminControl getAttribute $server name]
31            set ptype [$AdminControl getAttribute $server processType]
32            set pid [$AdminControl getAttribute $server pid]
33            set state [$AdminControl getAttribute $server state]
34            set jvm [$AdminControl queryNames type=JVM,
35            cell=$cname,node=$nname,process=$sname,*]
36            set osname [$AdminControl invoke $jvm getProperty os.name]
37            puts " $sname ($ptype) has pid
38            $pid; state: $state; on $osname"
39            #-----
40            # line 37-42 find the applications
41            running on this server and
42            # display the application name.
43            #-----
44            set apps [$AdminControl queryNames type=
45            Application,cell=$cname,node=$nname,process=$sname,*]
46            puts " Number of applications running
47            on $sname: [llength $apps]"
48            foreach app $apps {
49                set aname [$AdminControl getAttribute $app name]
50                puts " $aname"
```

```

42     }
43     puts "-----"
44     puts ""
45
46     }
47 }
48 }

```

- **5.1+** 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
17         # belonging to the cell
18         # and node, and process them one at a time
19         #-----
20         cname = AdminConfig.showAttribute(cell, 'name')
21         nname = AdminConfig.showAttribute(node, 'name')
22         servs = AdminControl.queryNames('type=Server,cell=' +
23         + cname + ',node=' + nname + ',*').split(lineSeparator)
24         print "Number of running servers on node
25         " + nname + ": %s \n" % (len(servs))
26         for server in servs:
27             #-----
28             # lines 28-34 get some attributes from
29             # the server to display;
30             # invoke an operation on the server JVM
31             # to display a property.
32             #-----
33             sname = AdminControl.getAttribute(server, 'name')
34             ptype = AdminControl.getAttribute(server, 'processType')
35             pid = AdminControl.getAttribute(server, 'pid')
36             state = AdminControl.getAttribute(server, 'state')
37             jvm = AdminControl.queryNames('type=JVM,cell=' +
38             + cname + ',node=' + nname + ',process=' + sname + ',*')
39             osname = AdminControl.invoke
40             (jvm, 'getProperty', 'os.name')
41             print " " + sname + " " + ptype + " has pid " +
42             pid + "; state: " + state + "; on " +
43             osname + "\n"
44
45         #-----
46         # line 40-45 find the applications
47         # running on this server and
48         # display the application name.
49         #-----
50         apps = AdminControl.queryNames('type=Application,cell=' +
51         + Cname + ',node=' + nname + ',process=' +
52         + sname + ',*').split(lineSeparator)
53         print "Number of applications running on "
54         + sname + ": %s \n" % (len(apps))
55         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
-----

```

Example: Starting an application using wsadmin

The following example starts an application:

- Identify the application manager MBean for the server where the application resides and assign it the appManager variable.

Using Jacl:

```

set appManager [$AdminControl queryNames cell=mycell,
node=mynode,type=ApplicationManager,process=server1,*]

```

5.1+ Using Jython:

```

appManager = AdminControl.queryNames('cell=mycell,node=
mynode,type=ApplicationManager,process=server1,*')
print appManager

```

This command returns the application manager MBean.

Example output:

```

WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=
ApplicationManager,type=ApplicationManager,
node=mynode,process=server1

```

- Start the application.

Using Jacl:

```

$AdminControl invoke $appManager startApplication myApplication

```

5.1+ Using Jython:

```

AdminControl.invoke(appManager, 'startApplication', 'myApplication')

```

This command invokes the startApplication operation on the MBean, passing in the application name to start.

Example: Stopping running applications on a server using wsadmin

The following example stops all running applications on a server:

- 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,*]
```

5.1+ Using Jython:

```
appManager = AdminControl.queryNames('cell=mycell,
node=mynode,type=ApplicationManager,process=server1,*')
print appManager
```

This command returns the application manager MBean.

Example output:

```
WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=
ApplicationManager,type=ApplicationManager,
node=mynode,process=server1
```

- 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,*]
```

5.1+ 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
```

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
```

- Stop all the running applications.

Using Jacl:

```
foreach app $apps {
    set appName [AdminControl getAttribute $app name]
    AdminControl invoke $appManager stopApplication $appName
}
```

5.1+ 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.

Example: Querying application state using wsadmin

The following examples queries for the presence of Application MBean to find out whether the application is running.

Using Jacl:

```
$AdminControl completeObjectName type=Application,  
name=myApplication,*
```

5.1+ Using Jython:

```
print AdminControl.completeObjectName  
( 'type=Application,name=myApplication,*' )
```

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
```

Example: Updating the Web server plug-in configuration files using wsadmin

This examples regenerates the Web server plug-in configuration file.

- Identify the Web server plug-in configuration file generator MBean and assign it to the pluginGen variable.

Using Jacl:

```
set pluginGen [ $AdminControl completeObjectName  
type=PluginCfgGenerator,* ]
```

5.1+ Using Jython:

```
pluginGen = AdminControl.completeObjectName  
( 'type=PluginCfgGenerator,*' )  
print pluginGen
```

Example output:

```
WebSphere:cell=pongoNetwork,name=PluginCfgGenerator,  
mbeanIdentifier=PluginCfgGenerator,type=PluginCfgGenerator,  
node=pongoManager,process=dmgr
```

- Generate the updated plug-in configuration file.

Using Jacl:

```
$AdminControl invoke $pluginGen generate "/WebSphere/  
DeploymentManager /WebSphere/DeploymentManager/config  
mycell null null plugin-cfg.xml"
```

5.1+ Using Jython:

```
AdminControl.invoke(pluginGen, 'generate', "/WebSphere/  
DeploymentManager /WebSphere/DeploymentManager/  
config mycell null null plugin-cfg.xml")
```

This example command assumes a Windows system install. It invokes the generate operation on the MBean, passing in the install root directory, configuration root directory, cell name, node name, server name, and output file name. To pass in null as the value of an argument, enter null as given in the example. This is provided for operation that allows null as the value of its argument and processes null differently from an empty string. In this example, both node and server are set to null. The generate operation generates plug-in configuration for all the nodes and servers resided in the cell. The output file plugin-cfg.xml is created in the config root directory.

You can modify this example command to generate plug-in configuration for a particular node or server by specifying the node and server names.

Managing applications with scripting

Application management scripts use the AdminApp object to manage applications in the application server configuration. You can use the AdminApp object to install and uninstall applications, list installed applications, edit application configurations and obtain help. It is important to save application configuration changes because the application configuration information is part of the server configuration.

1. Decide how you want to execute the script. If you want to run the script immediately from the command line, enter it surrounded by quotes as a parameter to the **wsadmin -c** command. To save the script for repeated use, compose it in a file and execute it with the **wsadmin -f** command. If you want to compose and run the script interactively, issue the **wsadmin** command without the **-c** or **-f** flags. For more information about executing scripts, see *Launching scripting clients*.
2. Write an AdminApp script command statement to perform a task, for example:

Using Jacl:

```
$AdminApp command
```

5.1+ Using Jython:

```
AdminApp.command()
```

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Installing applications with the wsadmin tool

Steps for using the AdminApp object commands to install an application into the run time follow:

1. Invoke the AdminApp 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:

The following command uses the EAR file and the command option information to install the application:

Using Jacl:

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

5.1+ Using Jython list:

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

5.1+ Using Jython string:

```
AdminApp.install('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
<i>MyStuff/application1.ear</i>	is the name of the application to install
server	is an installation option
<i>serv2</i>	is the value of the server option

The following command changes the application information by prompting you through a series of installation tasks:

Using Jacl:

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

5.1+ Using Jython:

```
AdminApp.installInteractive('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
<i>MyStuff/application1.ear</i>	is the name of the application to install

In a Network Deployment environment only, the following command uses the EAR file and the command option information to install the application on a cluster:

Using Jacl:

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

5.1+ Using Jython list:

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

5.1+ Using Jython string:

```
AdminApp.install('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
<i>MyStuff/application1.ear</i>	is the name of the application to install
cluster	is an installation option
<i>cluster1</i>	is the value of the server option

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Installing stand-alone Java archive and Web archive files with the wsadmin tool

Use the AdminApp object commands to install Java archive (JAR) and Web archive (WAR) files. The archive file must end in .jar or .war for the wsadmin tool to install. The wsadmin tool uses these extensions to figure out the archive type.

1. Invoke the AdminApp object commands interactively, in a script, or use `wsadmin -c` from an operating system command prompt.

2. Issue one of the following commands:

The following command uses the EAR file and the command option information to install the application:

Using Jacl:

```
$AdminApp install /MyStuff/mymodule1.jar {-server serv2}
```

5.1+ Using Jython list:

```
AdminApp.install('/MyStuff/mymodule1.jar', ['-server serv2'])
```

5.1+ Using Jython string:

```
AdminApp.install('/MyStuff/mymodule1.jar', ['-server' 'serv2'])
```

where:

\$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object supporting application management
install	is an AdminApp command
/MyStuff/mymodule1.jar	is the name of the application that will be installed
server	is an installation option
serv2	is the value of the server option

The following command allows you to change the application information by prompting you through a series of installation tasks:

Using Jacl:

```
$AdminApp installInteractive /MyStuff/mymodule1.jar
```

5.1+ Using Jython:

```
AdminApp.installInteractive('/MyStuff/mymodule1.jar')
```

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/mymodule1.jar	is the name of the application that will be installed

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Listing applications with the wsadmin tool

Use the AdminApp object commands to create a list of installed applications.

1. Invoke the AdminApp object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Query the configuration and create a list, by issuing the following command:

Using Jacl:

```
$AdminApp list
```

5.1+ 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

The following is example output:

```
DefaultApplication
SampleApp
app1serv2
```

Editing application configurations with the wsadmin tool

1. Invoke the AdminApp 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:

The following command uses the installed application and the command option information to edit the application:

Using Jacl:

```
$AdminApp edit app1 {options}
```

5.1+ Using Jython list:

```
AdminApp.edit('app1', ['options'])
```

5.1+ Using Jython string:

```
AdminApp.edit('app1', '[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
<i>app1</i>	is the name of the application to edit
{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 app1
```

5.1+ Using Jython:

```
AdminApp.editInteractive('app1')
```

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
<i>app1</i>	is the name of the application to edit

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

Uninstalling applications with the wsadmin tool

1. Invoke the AdminApp object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.

2. Issue the following command:

Using Jacl:

```
$AdminApp uninstall application1
```

5.1+ 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
<i>application1</i>	is the name of the application to uninstall

Note: Specify the name of the application you want to uninstall, not the name of the Enterprise ARchive (EAR) file.

3. Save the configuration changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ 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.

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.

Application management examples with wsadmin

There are examples that illustrate how to manage applications using wsadmin. Use these examples to see how to install, identify, configure and uninstall applications and application modules with the AdminApp object. Basic knowledge of the syntax for the Jacl scripting language is helpful in order to understand and modify the examples.

5.1+ You can also download a set of sample scripts from IBM developerWorks. To download the sample scripts, see Sample scripts for WebSphere Application Server (<http://www-106.ibm.com/developerworks/websphere/library/samples/SampleScripts.html>).

Example: Listing the modules in an installed application

Use the AdminApp object **listModules** command to list the modules in an installed application. For example, invoke the following command interactively in a script, or use wsadmin -c from an operating system command prompt:

Using Jacl:

```
$AdminApp listModules DefaultApplication -server
```

5.1+ Using Jython:

```
print AdminApp.listModules('DefaultApplication')
```

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 ==

```

```

119     "true" || $sameNodeDiffServer == "true" {
        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 $modServerNotMatch ""
147 }
148 }
149
150
151 proc filterAndPrint {lists flag} {
152     global webList
153     global ejbList
154     set webExists "false"
155     set ejbExists "false"
156
157     #-----
158     # If list already exists, flag it so as not to print the title more than once
159     # and reset the list
160     #-----
161     if {$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 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

```

Related reference

“Example: Listing the modules in an installed application” on page 220

Example: Obtaining task information while installing applications

The **installInteractive** command of the AdminApp object prompts you through a series of tasks when you install an application. You are presented with the title of the task, a description of the task, and the current contents of the task that you can modify.

- Use the **install** command instead of the **installInteractive** command, provide updates for each task, but you must provide all of the information on the

command line. The task name specifies each task and the information you need to update the task. You can treat the task information as a two-dimensional array of string data. For example:

Using Jacl:

```
-taskname {{item1a item2a item3a} {item1b item2b item3b} ...}
```

5.1+ Using Jython:

```
["-taskname", [{"item1a", "item2a", "item3a"},  
["item1b", "item2b", "item3b"]]]
```

This example is a linear representation of rows and columns, where {item1a item2a item3a} represents the first row, and each row for the task name has three columns.

The number and type of the columns in this list depend on the task you specify.

- Obtain information about the data needed for each task using the **taskInfo** command of the AdminApp object. For example, there is a task called *MapWebModToVH* used to map Web modules to virtual hosts. To specify this task as part of the option string on the command line, enter the following:

Using Jacl:

```
-MapWebModToVH {"JavaMail Sample WebApp" mtcomps.war,WEB-INF/  
web.xml default_host}}
```

5.1+ Using Jython:

```
["-MapWebModtoVH", [{"JavaMail Sample WebApp",  
"mtcomps.war,WEB-INF/web.xml", default_host}]]
```

Example Jacl output:

```
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}
```

5.1+ Example Jython output:

```
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]'
```

Using the **taskInfo** command, you can see which of the items you can change for a task. Supply the columns for each row you modify, and the columns that you are not allowed to change must match one of the existing rows. In this case,

taskInfo tells you that there are three items in each row, called *webModule*, *uri*, and *virtualHost* and the current column values for every row.

- Obtain help while creating complex installation commands, by using a feature of the **installInteractive** command. Install the application interactively once and specify the updates that you need. Then look for message WASX7278I in the output log for the wsadmin tool. You can cut and paste the data in this message into a script, and modify it. For example:

```
WASX7278I: Generated command line: install /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
-novaildateApp -installed.ear.destination
/mylocation -distributeApp -nouseMetaDataFromBinary}
```

Example: Identifying supported tasks and options for an Enterprise Archive file

The AdminApp object **install** command supports a set of options and tasks. The following examples use the AdminApp object to obtain a list of supported tasks and options for an Enterprise Archive (EAR) file:

- To identify supported options and tasks, use the AdminApp object **options** command:

Using Jacl:

```
$AdminApp options c:/MyStuff/myapp1.ear
```

5.1+ Using Jython:

```
AdminApp.options('c:/MyStuff/myapp1.ear')
```

This command displays a list of tasks and options.

- To identify supported options only, use the following command:

Using Jacl:

```
$AdminApp options
```

5.1+ Using Jython:

```
AdminApp.options()
```

- To learn more about any of the tasks or options, use the AdminApp object **help** command. For example:

Using Jacl:

```
$AdminApp help deployejb
```

5.1+ Using Jython:

```
AdminApp.help('deployejb')
```

Example: Configuring applications for enterprise bean modules using the wsadmin tool

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 enterprise bean modules for all the JARs in the application.

- Get the deployment object for the application and assign it to the deployments variable:

Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:myApp/]
```

5.1+ Using Jython:

```
deployments = AdminConfig.getid('/Deployment:myApp/')
print deployments
```

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/
myApp:deployment.xml#Deployment_1)
```

- Get all the modules in the application and assign it to the modules variable:

Using Jacl:

```
set deploymentObject [$AdminConfig showAttribute
$deployments deployedObject]
set modules [$AdminConfig showAttribute
$deploymentObject 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#EJBModuleDeployment_2)
```

5.1+ Using Jython:

```
deploymentObject = AdminConfig.showAttribute
(deployments, 'deployedObject')
print modules = AdminConfig.showAttribute
(deploymentObject, 'modules')
```

5.1+ 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)]
```

- Create an enterprise bean module configuration object for each JAR and set the timeout attribute:

Using Jacl:

```
foreach module $modules {
    if ([regexp EJBModuleDeployment $module] == 1) {
        $AdminConfig create EJBModuleConfiguration $module
        {{name myejbModuleConfig} {description "EJB Module Config post
created"} {enterpriseBeanConfigs:StatefulSessionBeanConfig
{{{ejbName myejb} {timeout 10000}}}}
    }
}
```

5.1+ Using Jython:

```
arrayModules = modules[1:len(modules)-1].split(" ")
for module in arrayModules:
    if module.find('EJBModuleDeployment') != -1:
        AdminConfig.create('EJBModuleConfiguration', module,
        [['name', 'myejbModuleConfig'], ['description',
"EJB Module Config post created"], ['enterpriseBeanConfigs:
StatefulSessionBeanConfig', [[['ejbName', 'myejb'],
['timeout', 10000]]]])
```

You can modify this example to set other attributes for the enterprise bean module configuration.

Example output:

```
myejbModuleConfig(cells/mycell/applications/myApp.ear/  
deployments/myApp:deployment.xml#EJBModuleConfiguration_1)
```

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring applications for session management using the wsadmin tool

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 the application.

- Identify the deployment configuration object for the application and assign it to the deployment variable:

Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:myApp/]
```

5.1+ Using Jython:

```
deployment = AdminConfig.getid('/Deployment:myApp/')  
print deployment
```

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/  
myApp:deployment.xml#Deployment_1)
```

- Retrieve the application deployment and assign it to the appDeploy variable:

Using Jacl:

```
set appDeploy [$AdminConfig showAttribute  
$deployment deployedObject]
```

5.1+ Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployment, 'deployedObject')  
print appDeploy
```

Example output:

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

- To obtain a list of attributes you can set for session manager, use the attributes command:

Using Jacl:

```
$AdminConfig attributes SessionManager
```

5.1+ Using Jython:

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

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"
```

- Set up the attributes for the session manager:

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]
```

5.1+ Using Jython:

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

This 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 TuningParams object types. To list the attributes for those object types, use the attribute command in AdminConfig object.

Example output using Jacl:

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

5.1+ Example output using Jython:

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

- Create the session manager for the application:

Using Jacl:

```
$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr]
```

5.1+ Using Jython:

```
print AdminConfig.create('ApplicationConfig', appDeploy, sessionMgr)
```

Example output:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

Example: Configuring applications for session management in Web modules using the wsadmin tool

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.

- Identify the deployment configuration object for the application and assign it to the deployment variable:

Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:myApp/]
```

5.1+ Using Jython:

```
deployment = AdminConfig.getid('/Deployment:myApp/')
print deployment
```

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/
myApp:deployment.xml#Deployment_1)
```

- Get all the modules in the application and assign it to the modules variable:

Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployment deployedObject]
set modules [$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)
```

5.1+ Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployment, 'deployedObject')
modules = AdminConfig.showAttribute(appDeploy, 'modules')
print modules
```

5.1+ 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)]
```

- To obtain a list of attributes you can set for session manager, use the attributes command.:

Using Jacl:

```
$AdminConfig attributes SessionManager
```

5.1+ Using Jython:

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

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"
```

- Set up the attributes for session manager:

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 tuningParamsDetailList [list $attr5]
set tuningParamsList [list tuningParams $tuningParamsDetailList]
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]

```

5.1+ Using Jython:

```

attr0 = ['enable', 'true']
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
attr4 = ['enableCookies', 'true']
attr5 = ['invalidationTimeout', 45]
tuningParamsDetailList = [attr5]
tuningParamsList = ['tuningParams', tuningParamsDetailList]
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])

```

This 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 attribute command in AdminConfig object.

Example output using Jacl:

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

5.1+ Example output using Jython:

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

- Set up the attributes for Web module:

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}}}}
```

5.1+ Using Jython:

```
nameAttr = ['name', 'myWebModuleConfig']
descAttr = ['description', "Web Module config post create"]
webAttrs = [nameAttr, descAttr, sessionMgr]
```

5.1+ Example output:

```
[[name, myWebModuleConfig], [description,
"Web Module config post create"],
[sessionManagement, [[enableSecurityIntegration, true],
[maxWaitTime, 30],
[sessionPersistenceMode, NONE], [enabled, true]]]]
```

- Create the session manager for each Web module in the application:

Using Jacl:

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

5.1+ Using Jython:

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

You can modify this example to set other attributes of session manager in Web module configuration.

Example output:

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

- Save the changes with the following command:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```


Example: Exporting applications using the wsadmin tool

Exporting applications enables you to back them up and preserve their binding information. You can export your applications before you update installed applications or before you migrate to a different version of the WebSphere Application Server product.

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

Using Jacl:

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

5.1+ Using Jython:

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

where *app1* is the name of the application that will be exported and */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 /mystuff
```

5.1+ Using Jython:

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

where *app1* is the name of the application whose DDL files will be exported and *C:/mystuff* is the name of the directory where the DDL files export from the application.

Example: Configuring a shared library for an application

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.

- Identify the shared library and assign it to the library variable.
 - To create a new shared library, perform the following steps:
 1. Identify the node and assign it to a variable, for example:

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

```
mynode(cells/mycell/nodes/mynode:node.xml#Node_1)
```

2. Create the shared library in the node, for example:

Using Jacl:

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

5.1+ Using Jython:

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

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode:libraries.xml#Library_1)
```

This example creates a new shared library in the node scope. You can modify it to use the cell or server scope.

- To use an existing shared library, issue the following command:

Using Jacl:

```
set library [$AdminConfig getid /Library:mySharedLibrary/]
```

5.1+ Using Jython:

```
library = AdminConfig.getid('/Library:mySharedLibrary/')  
print library
```

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode:libraries.xml#Library_1)
```

- Identify the deployment configuration object for the application and assign it to the deployment variable:

Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:myApp/]
```

5.1+ Using Jython:

```
deployment = AdminConfig.getid('/Deployment:myApp/')  
print deployment
```

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/  
myApp:deployment.xml#Deployment_1)
```

- Retrieve the application deployment and assign it to the appDeploy variable:

Using Jacl:

```
set appDeploy [$AdminConfig showAttribute  
$deployment deployedObject]
```

5.1+ Using Jython:

```
appDeploy = AdminConfig.showAttribute  
(deployment, 'deployedObject')  
print appDeploy
```

Example output:

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

- Identify the class loader in the application deployment and assign it to the classLoader variable:

Using Jacl:

```
set classLoader [$AdminConfig showAttribute  
$appDeploy classloader]
```

5.1+ Using Jython:

```
classLoader = AdminConfig.showAttribute  
(appDeploy, 'classloader')  
print classLoader
```

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/  
myApp:deployment.xml#Classloader_1)
```

- Associate the shared library in the application through the class loader:

Using Jacl:

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

5.1+ Using Jython:

```
print AdminConfig.create('LibraryRef', classLoader,
[['libraryName', 'MyshareLibrary'],
['sharedClassLoader', 'true']])
```

Example output:

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

- Save the changes:

Using Jacl:

```
$AdminConfig save
```

5.1+ Using Jython:

```
AdminConfig.save()
```

wsadmin scripting environment

The wsadmin tool contains facilities so that you can manage and customize the scripting environment. You can make temporary alterations to the scripting environment with the following wsadmin command options:

- `-profile` - Use this option to run one or more script files after you start the scripting tool.

Profile files are wsadmin scripts that initialize variables and define functions for the mainline scripts executed with the wsadmin tool. You can specify multiple profile options. They are executed in the order listed.

- `-p` - Use this option to specify scripting properties defined in a file.

Edit one or more properties files to make more persistent alterations to the scripting environment. The wsadmin tool loads the following levels of properties files:

- The properties in the `$WAS_ROOT/properties/wsadmin.properties` file.
- The properties in the `$user_home/wsadmin.properties` file.
- The properties indicated by the `WSADMIN_PROPERTIES` file.
- Any properties files specified on the command line.

The properties files load in this order. The properties file loaded last takes precedence over the one loaded before it.

The properties files include specifications for the connection type, the port and host used when attempting a connection, the location where trace and logging output are directed, the temporary directory to access files while installing applications and modules, class path information to append to the list of paths to search for classes and resources, and so on.

- `-wsadmin_classpath` - Use this option to add class path information to the wsadmin class path.
- `-conntype` - Use this option to specify the type of connection between the scripting client and the server.
- `-javaoption` - Use this option to pass Java standard or non-standard options to start the scripting tool.

wsadmin traces

The default properties file, `wsadmin.properties`, specifies that the tracing and logging information goes to the `wsadmin.traceout` file in the WebSphere logs directory.

It is recommended that trace output go to this or some other file. In the event of a script problem, you can examine this file for errors, or forward the file to IBM

Support if necessary. The wsadmin tool also creates a log entry for each command you issue interactively or with the -c option, and logs the script names that the -f option invokes. If the com.ibm.ws.scripting.traceString property is set in the properties file, diagnostic information also logs to this file. If the com.ibm.ws.scripting.traceFile property is not set in the properties file, this information goes to the console.

You can turn on traces of the WebSphere Application Server code running inside the scripting process by either specifying the com.ibm.ws.scripting.traceString property, or by using the AdminControl object trace method. If IBM Support personnel direct you to turn on such a trace, the output also goes to the file specified by the com.ibm.ws.scripting.traceFile property, or to the console, if that property is not in effect. You should use the trace command and the traceString property for setting up client traces only.

Tracing operations with the wsadmin tool

1. Invoke the AdminControl object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.
2. Enable tracing with the following command:

Using Jacl:

```
$AdminControl trace com.ibm.*=all=enabled
```

5.1+ 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
```

5.1+ 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 editing the wsadmin.properties file. The

property `com.ibm.ws.scripting.traceString` is read by the launcher during initialization. If it has a value, the value is used to set the trace.

A related property, `com.ibm.ws.scripting.traceFile`, designates a file to receive all trace and logging information. The `wsadmin.properties` file contains a value for this property. Run the `wsadmin` tool with a value set for this property. It is possible to run without this property set, where all logging and tracing goes to the administrative console.

Profiles and scripting

Scripting provides the capability to customize the environment in which scripts run by using the profile script. You can specify a profile in the following ways:

- Specify the **-profile** command option with `wsadmin`. You can specify more than one profile with the use of the multiple **-profile** option. The profile is invoked in the order given. An example on the Windows system follows:

Using Jacl:

```
wsadmin.sh -profile \myprofile1.jacl -profile \myprofile2.jacl
myprofile1.jacl is run before myprofile2.jacl
```

5.1+ Using Jython:

```
wsadmin.sh -lang jython -profile \myprofile1.py -profile \myprofile2.py
myprofile1.py is run before myprofile2.py
```

- Specify the profile scripts using the `com.ibm.ws.scripting.profiles` property in the properties file. You can specify multiple profiles by separating each profile script with a `;`. The profiles are invoked in the order given. An example of this property in the Windows system follows:

Using Jacl:

```
com.ibm.ws.scripting.profiles=/myprofile1.jacl;/myprofile2.jacl
```

5.1+ Using Jython:

```
com.ibm.ws.scripting.profiles=/myprofile1.py;/myprofile2.py
```

If profile is set in both the **-profile** option and as a property in the properties file, the profiles listed in the property file are invoked before the profiles in the command option.

If profile is specified, the profile is run when the scripting process starts. Any command specified with the **-c** command option and script file specified with the **-f** command option runs after the profiles are executed. In this way, the command and script file can use anything set up by the profiles. If the scripting process brings up an interactive session, then any procedures and variables defined in the profiles are available to the interactive session.

Properties used by scripted administration

Specifies the Java properties used by scripting administration.

There are three levels of default property files that load before any property file specified on the command line. The first level represents an installation default, located in the WebSphere Application Server properties directory called `wsadmin.properties`. The second level represents a user default, and is located in the Java user.home property just as `.wscprc` was in the WebSphere Application Server V4.0. This properties file is also called `wsadmin.properties`. The third level is a properties file 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 present on the command line. These three levels of property files load in the order that they are specified. The properties file 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 executing scripts. The `wsadmin.properties` file specifies Jacl as the scripting language.

5.1+ The supported scripting languages are Jacl and Jython. Other scripting languages that the Bean Scripting Framework (BSF) supports might work, but have not been tested.

com.ibm.ws.scripting.traceString

Turns on tracing for the scripting process. Tracing turned off is the default.

com.ibm.ws.scripting.traceFile

Determines where trace and log output is directed. The `wsadmin.properties` file specifies the `wsadmin.traceout` file located in the WebSphere Application Server properties directory 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 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` located in the WebSphere Application Server logs directory.

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 message WASX7207W 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 profiles to run automatically before running user commands, scripts, or an interactive shell.

5.1 + 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, `wsadmin` will perform conversion to change the profile names specified in this property to use file extension that matches the language specified. Use the default to make security configuration easier.

Java Management Extensions connectors

Use this page to view and change the configuration for Java Management Extensions (JMX) connectors.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers > *server_name* > Administration Services > JMX Connectors**
- **Servers > JMS Servers > *server_name* > Administration Services > JMX Connectors**

Java Management Extensions (JMX) connectors communicate with WebSphere Application Server when you invoke a scripting process. There is no default for the type and parameters of a connector. The `wsadmin.properties` file specifies the Simple Object Access Protocol (SOAP) connector and an appropriate port number. You can also use the Remote Method Invocation (RMI) connector.

Use one of the following methods to select the connector type and attributes:

- Specify properties in a properties file.
- Indicate options on the command line.

On the z/OS platform, both the SOAP connector and the RMI connector connect to a server control region. WebSphere for z/OS internally routes MBean requests from a server control region to its servant regions as appropriate.

Type

Specifies the type of the JMX connector.

Data type

Enum

Default
Range

SOAPConnector
SOAPConnector
For JMX connections using Simple Object Access Protocol (SOAP).
RMIConnector
For JMX connections using Remote Method Invocation (RMI).
HTTPConnector
For JMX connections using HTTP.
JMSConnector
For JMX connections using Java Messaging Service (JMS).

JMX connector settings

Use this page to view the configuration for a Java Management Extensions (JMX) connector.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers > *server_name* > Administration Services > JMX Connectors > *connector_type***
- **Servers > JMS Servers > *server_name* > Administration Services > JMX Connectors > *connector_type***

On the z/OS platform, both the SOAP connector and the RMI connector connect to a server control region. WebSphere for z/OS internally routes MBean requests from a server control region to its servant regions as appropriate.

Type:

Specifies the type of the JMX connector.

Data type
Default
Range

Enum
SOAPConnector
SOAPConnector
For JMX connections using Simple Object Access Protocol (SOAP).
RMIConnector
For JMX connections using Remote Method Invocation (RMI).
HTTPConnector
For JMX connections using HTTP.
JMSConnector
For JMX connections using Java Messaging Service (JMS).

Security and scripting

- Enabling and disabling security:

The wsadmin tool has two security related profiles by default that make security configuration easier. These profiles set up procedures that you can call to enable and disable security. The available procedures are:

securityon	turns global security on using LocalOS security
securityoff	turns global security off

LTPA_LDAPSecurityOn	turns LTPA/LDAP global security on using the LDAP user registry
LTPA_LDAPSecurityOff	turns LTPA/LDAP global security off

Using Jacl, enter the **securityon help** command or **LTPA_LDAPSecurityOn help** command to find out the parameters required for these procedures. For the procedures that turn security off, no parameters are required.

5.1+ Using Jython, enter the **securityon('help')** command or **LTPA_LDAPSecurityOn('help')** command to find out the parameters required for these procedures. For the procedures that turn security off, no parameters are required.

- Supplying user and password information:

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

You can specify user and password information on a wsadmin command line or the `sas.client.props` file located in the properties directory.

Use the **-user** and **-password** command options on the wsadmin tool to specify the user and password information.

The properties file updates required for running in secure mode will depend on whether a Remote Method Invocation (RMI) or Simple Object Access Protocol (SOAP) connector is being used to connect.

If you are using 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=
```

Change the value of the following property from prompt to properties:

```
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 will appear to hang.

If you are using a Simple Object Access Protocol (SOAP) connector, set the following properties in the `soap.client.props` file with the appropriate values:

```
com.ibm.SOAP.loginUserId=
com.ibm.SOAP.loginPassword=
com.ibm.SOAP.securityEnabled=true
```

There is no corresponding `com.ibm.SOAP.loginSource` property for a SOAP connector.

If you specify user and password information on a command line and in the properties file, the command line information will override the information in the properties file.

Scripting management examples with wsadmin

There are examples that illustrate how to customize the scripting environment using wsadmin. Basic knowledge of the syntax for the Jacl scripting language is helpful in order to understand and modify the examples.

Example: Using the wsadmin tool in a secure environment

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

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.

Example: Enabling and disabling LTPA_LDAP security with a profile using wsadmin

The following example calls the procedures set up by the default profile to enable and disable LTPA/LDAP security, based on single sign-on using LDAP user registry.

Enabling LTPA/LDAP global security:

- Use help to find out what arguments you need to provide:

Using Jacl:

```
LTPA_LDAPSecurityOn help
```

Example output:

```
Syntax: LTPA_LDAPSecurityOn server user password port domain
```

5.1+ Using Jython:

```
LTPA_LDAPSecurityOn('help')
```

5.1+ Example output:

```
Syntax: LTPA_LDAPSecurityOn(server, user, password, port, domain)
```

- Issue the call with the arguments provided to turn on LTPA/LDAP security:

Using Jacl:

```
LTPA_LDAPSecurityOn ldapServer1 user1 password1 660 ibm.com
```

5.1+ Using Jython:

```
LTPA_LDAPSecurityOn('ldapServer1', 'user1', 'password1', '660', 'ibm.com')
```

Example output:

PLEASE READ BELOW:

Done with LTPA/LDAP security turning on process, now you need to restart all the processes to make it affected. Then you can start using the client with SOAP or RMI connector.

- If you use the SOAP connector to connect to the server, you need to modify the soap.client.props file in your <install_root>/properties directory.

Update as below for SOAP connector:

```
com.ibm.SOAP.securityEnabled=true
com.ibm.SOAP.loginuserid=user1
com.ibm.SOAP.loginPassword=password1
```

- If you use the RMI connector to connect to the server, you are prompted to enter the user ID and the password. If you want to bypass the login process, you can modify sas.client.props file in your <install_root>/properties directory. Update as below for RMI connector:

```
com.ibm.CORBA.loginSource=properties
com.ibm.CORBA.loginuserid=user1
com.ibm.CORBA.loginPassword=password1
```

Disabling LTPA/LDAP global security:

- Issue the following call to turn off LTPA/LDAP global security

Using Jacl:

```
LTPA_LDAPSecurityOff
```

5.1+ Using Jython:

```
LTPA_LDAPSecurityOff()
```

Example output:

LTPA/LDAP security is off now but you need to restart all the processes to make it affected.

wsadmin tool performance tips

The following performance tips are for the wsadmin tool:

- When you launch a script using the wsadmin.bat or 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 execute 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 only load once.

The following example executes multiple application installation commands from a batch file:

Using Jacl:

```
wsadmin.sh -c "\AdminApp install /myApps//App1.ear {-appname app1}"
wsadmin.sh -c "\AdminApp install /myApps//App2.ear {-appname app12}"
wsadmin.sh -c "\AdminApp install /myApps//App3.ear {-appname app13}"
```

or

```
wsadmin.sh -c '$AdminApp install /myApps//App1.ear {-appname app1}'
wsadmin.sh -c '$AdminApp install /myApps//App2.ear {-appname app12}'
wsadmin.sh -c '$AdminApp install /myApps//App3.ear {-appname app13}'
```

5.1+ Using Jython:

```
wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App1.ear', '[-appname app1]')"
wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App2.ear', '[-appname app12]')"
wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App3.ear', '[-appname app13]')"
or
```

```
wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App1.ear', '[-appname app11]')'  
wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App2.ear', '[-appname app12]')'  
wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App3.ear', '[-appname app13]')
```

Or, for example, using Jacl, you can create the following file, *appinst.jacl*, that contains the commands:

```
$AdminApp install /myApps//App1.ear {-appname app11}  
$AdminApp install /myApps//App2.ear {-appname app12}  
$AdminApp install /myApps//App3.ear {-appname app13}
```

Then invoke this file using: `wsadmin -f appinst.jacl`

5.1 + Or, for example, using Jython, you can create the following file, *appinst.py*, that contains the commands:

```
AdminApp.install('/myApps/App1.ear', '[-appname app11]')  
AdminApp.install('/myApps/App2.ear', '[-appname app12]')  
AdminApp.install('/myApps/App3.ear', '[-appname app13]')
```

Then invoke this file using: `wsadmin.sh -lang jython -f appinst.py`

- Use the `AdminControl queryNames` and `completeObjectName` commands carefully with a large installation. For example, if there are only a few beans 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 actions that the `AdminControl` command performs on running MBeans, and actions that the `AdminConfig` command performs on configuration objects.

Chapter 8. Welcome to 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 commands available with the product. A subset of these are particular to system administration purposes.

Chapter 9. Managing 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*.

To manage using command line tools, perform the following steps:

1. Open a system command prompt.
2. Change to the bin directory.
3. Run 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
```
- 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. You can run this command from the `install_root/bin` directory of a WebSphere Application Server installation, or a Network Deployment installation.

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:

-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.

-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 file)
```

```
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. You can run this command from the `install_root/bin` directory of a WebSphere Application Server installation or a Network Deployment installation.

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.

-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/bin` directory of a Network Deployment installation.

Syntax

The command syntax is as follows:

```
startManager [options]
```

Parameters

The following options are available for the **startManager** command:

- 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.
- 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 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. You must run this command from the `install_root/bin` directory of a Network Deployment installation.

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.

-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)
```

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. You must run this command from the `install_root/bin` directory of a WebSphere Application Server installation.

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 <fileName>

Specifies the location of the log file to which information gets written.

-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)
```

START *appserver_proc_name* command

The **START** *appserver_proc_name* command reads the configuration file for the specified server process 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. You can run this command from the MVS console of a WebSphere Application Server installation, or a network deployment installation.

START *appserver_proc_name* parameters

Parameters

The parameters for the **START** *appserver_proc_name* command follow:

<appserver_proc_name>

Is the name of your start procedure for WebSphere Application Server for z/OS.

<server_short_name>

Is the short name of the J2EE server you are starting.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
START appserver_proc_name,JOBNAME=server_short_name,  
      ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

STOP *appserver_proc_name* command

The **STOP *appserver_proc_name*** 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** utility does not return control to the command line until the server completes shutting down. You can run this command from the MVS console of a WebSphere Application Server installation or a network deployment installation.

STOP *appserver_proc_name* parameters

Parameters

The parameters for the **STOP *appserver_proc_name*** command follow:

<appserver_proc_name>

Is the name of your WebSphere Application Server for z/OS stop procedure.

<server_short_name>

Is the short name of the J2EE server you are stopping.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
STOP appserver_proc_name,JOBNAME=server_short_name,  
    ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

START *dmgr_proc_name* command

The **START *dmgr_proc_name*** command reads the configuration file for the network deployment manager process and constructs a launch command. Depending on the options you specify, the **START *dmgr_proc_name*** 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 MVS console of a network deployment installation.

START <*dmgr_proc_name*> parameters

Parameters

The parameters for the **START *dmgr_proc_name*** command follow:

<dmgr_proc_name>

Is the name of your WebSphere Application Server for z/OS deployment manager start procedure.

<server_short_name>

Is the short name of the J2EE server the deployment manager will be managing.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
START dmgr_proc_name,JOBNAME=server_short_name,  
      ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

STOP *dmgr_proc_name* command

The **STOP *dmgr_proc_name*** 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** utility waits for the manager to complete shutdown before it returns control to the command line. You must run this command from the MVS console.

STOP *dmgr_proc_name* parameters

Parameters

The parameters for the **STOP *dmgr_proc_name*** command follow:

<dmgr_proc_name>

Is the name of your WebSphere Application Server for z/OS deployment manager stop procedure.

<server_short_name>

Is the short name of the J2EE server the deployment manager is managing.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
STOP dmgr_proc_name,JOBNAME=server_short_name,  
      ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

START *nodeagent_proc_name* command

The **START *nodeagent_proc_name*** command reads the configuration file for the node agent process, and constructs a launch command. Depending on the options that you specify, the **START *nodeagent_proc_name*** command creates a new Java Virtual Machine (JVM) API to run the agent process, or writes the launch command data to a file. You must run this command from the MVS console of a WebSphere Application Server installation.

START *nodeagent_proc_name* parameters

Parameters

The parameters for the **START** *nodeagent_proc_name* command follow:

<nodeagent_proc_name>

Is the name of your WebSphere Application Server for z/OS node agent start procedure.

<server_short_name>

Is the short name of the J2EE server you want this node agent to manage.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
START nodeagent_proc_name,JOBNAME=server_short_name,  
      ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

STOP *nodeagent_proc_name* command

The **STOP** *nodeagent_proc_name* command reads the configuration file for the network deployment node agent process. It sends a Java Management Extensions (JMX) command to the node agent telling it to shut down. By default, the **stopNode** utility waits for the node agent to complete shutdown before it returns control to the command line. You must run this command from the MVS console of a WebSphere Application Server installation.

STOP *nodeagent_proc_name* parameters

Parameters

The parameters for the **STOP** *nodeagent_proc_name* command follow:

<nodeagent_proc_name>

Is the name of your WebSphere Application Server for z/OS node agent stop procedure.

<server_short_name>

Is the short name of the J2EE server this node agent is managing.

<cell_short_name>

Is the short name of the cell containing this J2EE server.

<node_short_name>

Is the short name of the of the node containing this J2EE server.

The following example demonstrates correct syntax:

```
STOP nodeagent_proc_name,JOBNAME=server_short_name,  
      ENV=cell_short_name.node_short_name.  
server_short_name
```

Note: This command must be entered on a single line. It is split here for display purposes.

addNode command

The **addNode** command incorporates a WebSphere Application Server installation into a cell. You must run this command from the `install_root/bin` directory of a WebSphere Application Server installation. 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. 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.

Syntax

The command syntax is as follows:

```
addNode <deploymgr host> <deploymgr port> [options]
```

The `<deploymgr host>` argument is required. 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 install the deployment manager using coexistence ports, the default SOAP port is 8889.

Parameters

The following options are available for the **addNode** command:

-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.

-replacelog

Replaces the log file instead of appending to the current log.

-trace

Generates trace information into a file for debugging purposes.

-replacelog

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.

-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 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. This option tells the **addNode** command to attempt to include applications from the new node. If the application already exists in the cell, a warning is printed and the application is not installed into the cell.

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
```

You have to use the `-includeApps` option to migrate all the applications to the new cell. 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.

-startingport <portNumber>

Supports the specification of a port number to use as the base port number for all node agent and Java Messaging Service (JMS) server 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 and JMS server port configured during the **addNode** command.

-help

Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
addNode testhost 8879
```

```
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.Example output:

```
D:\WebSphere\AppServer\bin>addnode <dmgr_host>
ADMU0116I: Tool information is being logged in file
          D:\WebSphere\AppServer\logs\addNode.log
ADMU0001I: Begin federation of node <node_name> with deployment manager at
          <dmgr_host>:8879.
ADMU0009I: Successfully connected to deployment manager server: <dmgr_host>:8879.
ADMU0505I: Servers found in configuration:
ADMU0506I: Server name: server1
ADMU2010I: Stopping all server processes for node <node_name>
ADMU0512I: Server server1 cannot be reached. It appears to be stopped.
ADMU0024I: Deleting the old backup directory.
ADMU0015I: Backing up the original cell repository.
ADMU0012I: Creating node agent configuration for node: <node_name>
ADMU0014I: Adding node <node_name> configuration to cell: <cell_name>
ADMU0016I: Synchronizing configuration between node and cell.
ADMU0018I: Launching node agent process for node: <node_name>
ADMU0020I: Reading configuration for node agent process: nodeagent
ADMU0022I: Node agent launched. Waiting for initialization status.
ADMU0030I: Node agent initialization completed successfully. Process ID is:
          2340
ADMU0523I: Creating queue manager for node <node_name> on server jmsserver
ADMU0525I: Details of queue manager creation can be seen in the file:
          createMQ.<node_name>_jmsserver.log
ADMU9990I:
ADMU0300I: Congratulations! Your node <node_name> has been successfully
          incorporated into the <cell_name> cell.
ADMU9990I:
ADMU0306I: Be aware:
ADMU0302I: Any cell-level documents from the stand-alone <node_name>
          configuration have not been migrated to the new cell.
ADMU0307I: You might want to:
ADMU0303I: Update the configuration on the <cell_name> deployment manager
          with values from the old cell-level documents.
ADMU9990I:
ADMU0306I: Be aware:
ADMU0304I: Because the -includeapps option was not specified, applications installed on
          the stand-alone node were not installed on the new cell.
ADMU0307I: You might want to:
ADMU0305I: Install applications onto the <cell_name> cell using the wsadmin
          $AdminApp command or the administrative console.
ADMU9990I:
ADMU0003I: Node <node_name> has been successfully federated.
```

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 in Fix Pack 2

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 port number settings in WebSphere Application Server versions.

Tips for using the **addNode** command:

- **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. You can run this command from the `install_root/bin` directory of a WebSphere Application Server installation or a Network Deployment installation.

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.

-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. You must run this command from the `install_root/bin` directory of a WebSphere Application Server installation.

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.

-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. You must run this command from the `install_root/bin` directory of a Network Deployment installation.

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.

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. Only use this command when you cannot run the node agent because the node configuration does not match the cell configuration. You must run this command from the `install_root/bin` directory of a WebSphere Application Server installation.

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.
- 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)
```

```
syncNode.sh testhost 8879
```

```
syncNode.sh deploymgr 8879 -trace (produces the syncNode.log file)
```

```
syncNode.sh 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. You can run this command from the `install_root/bin` directory of a WebSphere Application Server installation or a network deployment installation.

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.

-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. You can run this command from the `install_root/bin` directory of a WebSphere Application Server installation or a Network Deployment installation.

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:

-nostop

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.

-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.sh 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.sh WebSphereConfig_2003-04-22.zip -location /tmp -nostop
```

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.

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.

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

Purpose

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`. This file is located in the `bin` directory of the product installation root, and has the following command-line syntax:

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 [Configuring Web server plug-ins](#).

To regenerate the plug-in configuration, you can either use the Update Web Server Plug-in Configuration page in the administrative console, or issue the following commands:

```
WAS_HOME/AppServer/GenPluginCfg.sh
```

WAS_HOME is the root directory for your installation of IBM WebSphere Application Server.

Both methods for regenerating the plug-in configuration create a `plugin-cfg.xml` file in EBCDIC format, which is the proper format for execution in a z/OS environment.

Chapter 10. Managing using mvs command line tools

See the z/OS Internet Library ([http://www-ibm.com/servers/eserver/zseries/zos/bkserv](http://www.ibm.com/servers/eserver/zseries/zos/bkserv)) for information on how to use MVS operator commands.

Modify command

Use the **modify** command from the MVS console to dynamically modify WebSphere for z/OS operations.

You can use the modify command to display status of various server components and activities, including:

- Active controllers
- Trace settings
- Servants
- Sessions
- JVM Heap
- Java trace

f <server>, options

The first argument is required. The argument is either the name of the configuration directory of the server for which status is desired, or the **-all** keyword which requests status for all servers defined on the node.

Parameters

The options for the **modify** command follow:

CANCEL

Used to cancel a server. *server* refers to the server short name.

You can specify the following options:

ARMRESTART

Specify if you are using ARM and want ARM to restart the server after it terminates. If you don't specify the ARMRESTART option on the CANCEL parameter, ARM will not restart the server.

HELP Get help for the CANCEL syntax.

Note: You cannot use the CANCEL parameter to cancel a cluster from the MVS console. Instead, you must cancel each of the servers that make up the cluster.

TRACEALL=n

Use TRACEALL to establish a general trace level for the server.

Valid trace levels are 0 (none), 1 (exception), 2 (basic), and 3 (detailed tracing). Under normal conditions and in production, use 1 (exception).

Note: Be careful when using a level of 3 (detailed for all components) because it can potentially yield more data than can be handled reasonably.

TRACEBASIC=n

Specify the WebSphere for z/OS components for which you want to switch on a basic level of tracing.

This command has the ability to override a different tracing level established by TRACEALL for those components.

Note: Do not change this variable unless directed by IBM service personnel.

You can specify one or more of the following options for either TRACEBASIC or TRACEDetail:

0	RAS
1	Common Utilities
3	COMM
4	ORB
6	OTS
7	Shasta
9	OS/390 Wrappers
A	Daemon
E	Security
F	Externalization
J	JRAS (internal tracing-via direction from IBM support)
L	J2EE

TRACEDetail=n

Specify the WebSphere for z/OS components for which you want to switch on a detailed level of tracing.

This command activates the most detailed tracing for the specified WebSphere for z/OS components and overrides different settings inTRACEALL. The selected components (*n,...*) are identified by their component-ID (valid values are the same as for TRACEBASIC above). Subcomponents, specified by numbers, receive detailed traces. Other parts of WebSphere for z/OS receive tracing as specified on the TRACEALL variable.

Note: Do not change this variable unless directed by IBM service personnel.

TRACESPECIFIC=xyyyzzz

Specifies tracing overrides for specific WebSphere for z/OS trace points.

Trace points are specified by 8-digit, hexadecimal numbers. To specify more than one trace point, use parentheses and separate the numbers with commas. You can also specify an environment variable name by enclosing the name in single quotes. The value of the environment variable will be handled as if you had specified that value on TRACESPECIFIC.

Note: Do not use TRACESPECIFIC unless directed by IBM service personnel.

TRACE_EXCLUDE_SPECIFIC=xyyyzzz

Specifies trace points to exclude.

Trace points to exclude are specified by 8-digit, hexadecimal numbers. To specify more than one trace point, use parentheses and separate the numbers with commas. You can also specify an environment variable name by enclosing the name in single quotes. The value of the environment variable will be handled as if you had specified that value on TRACE_EXCLUDE_SPECIFIC. You can use TRACE_EXCLUDE_SPECIFIC as a mask to turn off otherwise-on

traces. For example, use the TRACESPECIFIC command to turn on tracing for a whole part and then use TRACE_EXCLUDE_SPECIFIC to turn off one trace within that part.

Note: Do not use TRACE_EXCLUDE_SPECIFIC unless directed by IBM service personnel.

TRACEINIT

Reset to the initial trace settings.

TRACENONE

Turns off all trace settings.

TRACETOSYSPRINT={YES | NO}

Select whether or not to send the trace to sysprint.

YES specifies to send the trace to sysprint and NO stops sending the trace to sysprint.

TRACETOTRCFILE={YES | NO}

Select whether or not to direct the trace to the TRCFILE DD card.

YES specifies to send the trace to the TRCFILE DD card and NO stops sending the trace to the TRCFILE DD card.

TRACEJAVA

Modify the Java trace string.

The java trace specification is used to control java tracing and conforms to the java trace specification rules. *=all=enabled means to enable all types of tracing for all registered trace components.

HELP

Display a list of all the keywords you may use with the modify command.

You can also use the HELP parameter after the CANCEL, and DISPLAY parameters to display lists of all the keywords you can use with either of these parameters.

DISPLAY | DISPLAY,

Displays the server's name, the system name where it is running, and the current code level.

You can specify the following options:

SERVERS

This command is directed to a server and displays the name, system name, and code level for each active server in the sysplex that is in the same cell.

SERVANTS

Displays a list of ASIDs of servants attached to the server against which you issued the display command.

TRACE

Display trace information for a server controller.

You can further modify this command with one of the following options:

SRS Display trace information for all servants, one at a time.

ALL Display trace information for the controller and all servants one at a time.

JAVA Display the Java trace string settings for a server controller. You can further modify this command with one of the following options:

SRS Display Java trace information for all servants, one at a time.

ALL Display Java trace information for the controller and all servants one at a time.

HELP Display a list of all the keywords you may use with the modify display trace java command.

HELP Display a list of all the keywords you may use with the modify display trace command.

JVMHEAP

Display the JVM heap information for a server controller.

You can further modify this command with one of the following options:

SRS Display the JVM heap information for all servants, one at a time.

ALL Display the JVM heap information for the controller and all servants, one at a time.

HELP Display a list of all the keywords you may use with the modify display javaheap command.

SESSIONS

Display session information for the server.

You can further modify this command with one of the following options:

LISTENERS

Display the listening port numbers for each protocol. This is actually the default, so the *f server,sessions* and the *f server,sessions,listeners* commands would have the same outcome.

SERVER

Display the number of sessions in use for each protocol on the server.

You can further modify this command with one of the following options:

TCPIIOP

Display the number of TCP/IP IIOP sessions active on the server.

You can further modify this command with one of the following options:

LIST List the server session information for the TCP/IP IIOP protocol.

HELP Display a list of all the keywords you may use with the modify display session server tcpipiiop command.

Note: The LIST and HELP parameters are also available with the other session protocol display commands.

LOCALIOP

Display the number of LOCALIOP sessions active on the server.

SSLIOP

Display the number of SSLIOP sessions active on the server.

HTTP Display the number of HTTP sessions active on the server.

HTTPS

Display the number of HTTPS sessions active on the server.

HELP Display a list of all the keywords you may use with the modify display session server command.

HELP Display help for the modify display sessions command.

HELP Display a list of all the keywords you may use with the modify command.

Example: Canceling application clusters and servers with the modify command

Before you begin: You cannot cancel a cluster from the MVS console. Instead, you must cancel each of the servers that make up the cluster.

Example 1: The following command will cancel the *bbo5acr* server:

```
f bbo5acr, cancel
```

Example 2: The following command will cancel the *bbo5acr* server and instruct ARM to restart it after it terminates:

```
f bbo5acr, cancel, armrestart
```

Example: Establishing a general level of trace

To establish a general trace level for the server, use the following command:

```
f server, traceall=n
```

Valid trace levels are 0 (none), 1 (exception), 2 (basic), and 3 (detailed tracing). Under normal conditions and in production, use 1 (exception).

Example: The following command will turn on exception level tracing for the *bbo5acr* server:

```
f bbo5acr, traceall=1
```

Here is a sample display:

```
F BB05ACR, TRACEALL=1  
BB000211I MODIFY COMMAND TRACEALL=1 COMPLETED SUCCESSFULLY
```

Example: Setting basic and detailed trace levels

To specify the WebSphere for z/OS components for which you want to switch on a basic level of tracing, use the following command:

```
f server,tracebasic=(n,...)
```

Example 1: The following command will turn on a basic level of tracing for the Daemon component on server *bbo5acr*:

```
f bbo5acr,tracebasic=a
```

Here is a sample display:

```
F BB05ACR,TRACEBASIC=A  
BB000211I MODIFY COMMAND TRACEBASIC=A COMPLETED SUCCESSFULLY
```

To specify the WebSphere for z/OS components for which you want to switch on a detailed level of tracing use the following command:

```
f server,tracedetail=(n,...)
```

Example 2: The following command will turn on a detailed level of tracing for security on server *bbo5acr*:

```
f bbo5acr,tracedetail=e
```

Here is a sample display:

```
F BB05ACR,TRACEDetail=E  
BB000211I MODIFY COMMAND TRACEDetail=E COMPLETED SUCCESSFULLY
```

Example: Setting specific trace points

To set specific trace points, use the following command:

```
f server,tracespecific=n | (n,...)
```

Example 1: The following command will turn on the specific trace point 04006001:

```
f bbo5acr,tracespecific=04006001
```

Example 2: The following command will turn on the specific trace points 04006001 and 04006027:

```
f bbo5acr,tracespecific=(04006001,04006027)
```

Example 3: The following command will turn on the specific trace points set in the environment variable *tracepoints*:

```
f bbo5acr,tracespecific='tracepoints'
```

Example: Excluding specific trace points

To exclude specific trace points, use the following command:

```
f server,trace_exclude_specific=n | (n,...)
```

Example: The *tracespecific* command below turns on tracing for a whole part and then the *trace_exclude_specific* turns off tracing for the point 04006031 within that part.:

```
f bbo5acr,tracespecific=04006000
```

```
f bbo5acr,trace_exclude_specific=04006031
```

Example: Resetting to the initial trace settings

To reset to the initial trace settings use the following command:

```
f server,traceinit
```

Example: Turning off tracing

To turn off tracing, use the following command:

```
f server, tracenone
```

Example: Sending the trace to sysprint

To send the trace to sysprint, use the following command:

```
f server, tracetosysprint=yes
```

To stop sending the trace to sysprint, use the following command:

```
f server, tracetosysprint=no
```

Example: Displaying servants

To display servants, use the following command:

```
f server, display, servants
```

Example: The following command will display servants for the bbo5acr server:

```
f bbo5acr, display, servants
```

Here is a sample display:

```
F BB05ACR, DISPLAY, SERVANTS
BB000185I SERVER BB05SR4/BB0ASR4A HAS 1 SERVANT PROCESS (ASID: 0038x)
BB000188I END OF OUTPUT FOR COMMAND DISPLAY, SERVANTS
```

Example: Displaying trace settings and Java string trace settings

To display the trace settings for a server instance, use the following command:

```
f server, display, trace
```

Example 1: The following command will display trace settings for the bbo5acr server:

```
f bbo5acr, display, trace
```

Here is a sample display:

```
F BB05ACR, DISPLAY, TRACE
BB000224I TRACE INFORMATION FOR SERVER BB05SR4/BB0ASR4A/STC00047
BB000197I LOCATION = SYSPRINT BUFFER
BB000197I AGGREGATE TRACE LEVEL = 1
BB000197I EXCEPTION TRACING = RAS(0), Common Utilities(1), COMM(3),
ORB(4), OTS(6), Shasta(7), OS/390 Wrappers(9), Daemon(A), Security(E),
Externalization(F), JRAS(J), J2EE(L)
BB000197I BASIC TRACING =
BB000197I DETAILED TRACING =
BB000197I TRACE SPECIFIC = NONE SPECIFIED
BB000197I TRACE EXCLUDE SPECIFIC = NONE SPECIFIED
BB000225I TRACE INFORMATION FOR SERVER BB05SR4/BB0ASR4A/STC00047 COMPLETE
BB000188I END OF OUTPUT FOR COMMAND DISPLAY, TRACE
```

To display the Java trace string settings for a server, use the following command:

```
f server, display, trace, java
```

Example 2: The following command will display Java trace settings for the bbo5acr server:

```
f bbo5acr, display, trace, java
```

Here is a sample display:

```
F BB05ACR,DISPLAY,TRACE,JAVA
BB000196I TRACE INFORMATION FOR SERVER BB05SR4/BBOASR4A
BB0J0050I CTL(STC00047):*=all=disabled
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA
```

To display the Java trace information for all servants, one at a time.

```
f server,display,trace,java.srs
```

Example 3: The following command will display Java trace settings for the bbo5acr server:

```
f bbo5acr,display,trace,java,srs
```

Here is a sample display:

```
F BB05ACR,DISPLAY,TRACE,JAVA,SRS
BB000196I TRACE INFORMATION FOR SERVER BB05SR4/BBOASR4A
BB0J0050I SR(STC00048):*=all=disabled
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA,SRS
```

To display the Java trace information for the controller and all servants, one at a time.

```
f server,display,trace,java.all
```

Example 4: The following command will display Java trace settings for the controller and all servants:

```
f bbo5acr,display,trace,java,all
```

Here is a sample display:

```
F BB05ACR,DISPLAY,TRACE,JAVA,ALL
BB000196I TRACE INFORMATION FOR SERVER BB05SR4/BBOASR4A
BB0J0050I CTL(STC00047):*=all=disabled
BB0J0050I SR(STC00048):*=all=disabled
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA,ALL
```

Example: Displaying JVM heap information

To display the JVM heap information for a server controller, use the following command:

```
f server,display,jvmheap
```

Example 1: The following command will display JVM heap information for the bbo5acr server:

```
f bbo5acr,display,jvmheap
```

Here is a sample display:

```
F BB05ACR,DISPLAY,JVMHEAP
BB000201I JVM HEAP INFORMATION FOR SERVER BB05SR4/BBOASR4A/STC00047
BB000202I (STC00047) HEAP(MIDDLEWARE), COUNT(00000000), FREE STORAGE(
396FA70), TOTAL STORAGE( 7FFFA00)
BB000204I JVM HEAP INFORMATION FOR SERVER
BB05SR4/BBOASR4A/STC00047 COMPLETE
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP
```

To display the JVM heap information for all servants one at a time, use the following command:

```
f server,display,jvmheap,srs
```

Example 2: The following command will display JVM heap information for all servants in the bbo5acr server:

```
f bbo5acr,display,jvmheap,srs
```

Here is a sample display:

```
F BB05ACR,DISPLAY,JVMHEAP,SRS
+BB000201I JVM HEAP INFORMATION FOR SERVER BB05SR4/BB0ASR4A/STC00048
+BB000202I (STC00048) HEAP(MIDDLEWARE), COUNT(00000001), FREE STORAGE(
 25F4030), TOTAL STORAGE( 7FFFA00)
+BB000204I JVM HEAP INFORMATION FOR SERVER
  BB05SR4/BB0ASR4A/STC00048 COMPLETE
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP,SRS
```

To display the JVM heap information for the controller and all servants of a server one at a time, use the following command:

```
f server,display,jvmheap,all
```

Example 3: The following command will display JVM heap information for the controller and all servants in the bbo5acr server:

```
f bbo5acr,display,jvmheap,all
```

Here is a sample display:

```
F BB05ACR,DISPLAY,JVMHEAP,ALL
BB000201I JVM HEAP INFORMATION FOR SERVER BB05SR4/BB0ASR4A/STC00047
BB000202I (STC00047) HEAP(MIDDLEWARE), COUNT(00000000), FREE STORAGE(
396FA70), TOTAL STORAGE( 7FFFA00)
BB000204I JVM HEAP INFORMATION FOR SERVER
  BB05SR4/BB0ASR4A/STC00047 COMPLETE
+BB000201I JVM HEAP INFORMATION FOR SERVER BB05SR4/BB0ASR4A/STC00048
+BB000202I (STC00048) HEAP(MIDDLEWARE), COUNT(00000001), FREE STORAGE(
 25F4030), TOTAL STORAGE( 7FFFA00)
+BB000204I JVM HEAP INFORMATION FOR SERVER
  BB05SR4/BB0ASR4A/STC00048 COMPLETE
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP,ALL
```

Example: Displaying sessions

To display sessions, use the following command:

```
f server,
display,sessions
```

Example 1: Since "listeners" is the default, either of the following command will display session listeners for the bbo5acr server:

```
f bbo5acr,display,sessions
```

or

```
f bbo5acr,display,sessions,listeners
```

Here is a sample display:

```
F BB05ACR,DISPLAY,SESSIONS,LISTENERS
BB000189I PROTOCOLS AND LISTENING PORTS FOR SERVER BB05SR4/BB0ASR4A
BB000190I TCPIIOP      2809
BB000190I SSLIIOP      6000
BB000190I HTTP         9088
BB000190I HTTPS        443
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,SESSIONS,LISTENERS
```

To display server sessions counts, use the following command:

```
f server,  
display,sessions,server
```

Example 2: The following command will display the server session counts for all protocols used by the bbo5acr server:

```
f bbo5acr,display,sessions,server
```

Here is a sample display:

```
F BB05ACR,DISPLAY,SESSIONS,SERVER  
BB000191I SERVER SESSION COUNTS FOR PROTOCOLS USED BY SERVER  
BB05SR4/BB0ASR4A  
BB000192I TCPIIOP      2  
BB000192I LOCALIOP    0  
BB000192I SSLIOP      0  
BB000192I HTTP        0  
BB000192I HTTPS        0  
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,SESSIONS,SERVER
```

You can further refine the modify command to display, or list, the number of sessions used by a specific protocol. The available options are:

- tcpiiop
- localiop
- ssliop
- http
- https

In addition, you can also specify to display the results or list the results by leaving off the list modifier or including the list modifier, respectively.

Example 1: The following command will display the number of sessions using the TCP/IP IIOp protocol on the bbo5acr server:

```
f bbo5acr,display,sessions,server,tcpiiop
```

Here is a sample display:

```
F BB05ACR,DISPLAY,SESSIONS,SERVER,TCPIIOP  
BB000193I SERVER BB05SR4/BB0ASR4A HAS 2 TCPIIOP  SERVER SESSIONS  
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,SESSIONS,SERVER,TCPIIOP
```

Example: Displaying status of a server

To display the status of a server, use the following command:

```
f server,display
```

Example: The following command will display status for the bbo5acr server:

```
f bbo5acr,display
```

Here is a sample display:

```
F BB05ACR,DISPLAY  
BB000173I SERVER BB05SR4/BB0ASR4A ACTIVE ON SY1 AT LEVEL wd5xo03.  
BB000188I END OF OUTPUT FOR COMMAND DISPLAY
```

Example: Displaying status of clusters

To display the status of clusters in the active controller, use the following command:

```
f server,display,servers
```

Example: The following command will display status for the bbo5acr server:


```
f bbo5acr,display,servers
```

Here is a sample display:

```
F BB05ACR,DISPLAY,SERVERS
BB000182I SERVER                ASID  SYSTEM  LEVEL
BB000183I CBDAEMON/DAEMON01     31x  SY1    wd5xo03
BB000183I BB05SR4 /BBOASR4A     1F6x SY1    wd5xo03
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,SERVERS
```

Example: Getting help for the modify command

You can get syntax help for the various levels of the modify command.

Example 1: The following command will display a list of all the keywords you may use with the modify command:

```
f bbo5acr,help
```

Here is a sample display:

```
F BB05ACR,HELP
BB000178I THE COMMAND MODIFY MAY BE FOLLOWED BY ONE OF THE FOLLOWING KEYWORDS:
BB000179I CANCEL - CANCEL THIS CONTROL REGION
BB000179I TRACEALL - SET OVERALL TRACE LEVEL
BB000179I TRACEBASIC - SET BASIC TRACE COMPONENTS
BB000179I TRACEDetail - SET DETAILED TRACE COMPONENTS
BB000179I TRACESPECIFIC - SET SPECIFIC TRACE POINTS
BB000179I TRACEINIT - RESET TO INITIAL TRACE SETTINGS
BB000179I TRACENONE - TURN OFF ALL TRACING
BB000179I TRACETOSYSPRINT - SEND TRACE OUTPUT TO SYSPRINT (YES/NO)
BB000179I TRACETOTRCFILE - SEND TRACE OUTPUT TO TRCFILE DD CARD (YES/NO)
BB000179I DISPLAY - DISPLAY STATUS
BB000179I TRACE_EXCLUDE_SPECIFIC - EXCLUDE SPECIFIC TRACE POINTS
BB000179I TRACEJAVA - SET JAVA TRACE OPTIONS
```

Example 2: The following command will display a list of all the keywords you may use with the modify display command:

```
f bbo5acr,display,help
```

Here is a sample display:

```
F BB05ACR,DISPLAY,HELP
BB000178I THE COMMAND DISPLAY, MAY BE FOLLOWED BY ONE OF THE FOLLOWING KEYWORDS:
BB000179I SERVERS - DISPLAY ACTIVE CONTROL REGIONS
BB000179I SERVANTS - DISPLAY SERVANT PROCESSES OWNED BY THIS CONTROL PROCESS
BB000179I SESSIONS - DISPLAY INFORMATION ABOUT COMMUNICATIONS SESSIONS
BB000179I TRACE - DISPLAY INFORMATION ABOUT TRACE SETTINGS
BB000179I JVMHEAP - DISPLAY JVM HEAP STATISTICS
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,HELP
```

Example 3: The following command will display a list of all the keywords you may use with the modify display sessions command:

```
f bbo5acr,display,sessions,help
```

Here is a sample display:

```
F BB05ACR,DISPLAY,SESSIONS,HELP
BB000178I THE COMMAND DISPLAY,SESSIONS, MAY BE FOLLOWED BY ONE OF THE
FOLLOWING KEYWORDS:
BB000179I LISTENERS - DISPLAY LISTENER INFORMATION
BB000179I SERVER - DISPLAY SERVER SESSION INFORMATION
BB000188I END OF OUTPUT FOR COMMAND DISPLAY,SESSIONS,HELP
```

Example: Modifying the Java trace string

To modify the Java trace string, use the following command:

```
f server, tracejava='trace specification'
```

Example: The following command enables all types of tracing for all registered trace components:

```
f bbo5acr, tracejava='*=all=enabled'
```

Note: The single quotes are a required part of the syntax.

Display command

Examples

Use the following display command examples to monitor your WebSphere Application Server.

- Display active replies
- Display active address spaces
- Display the status of address spaces registered for automatic restart management
- Display units of work (transactions) for the Information Management System

Example: Displaying active replies

Displaying active replies from the MVS console allows you to observe system activity and determine if the system requires an operator response.

Issue the following command to display (list) all active replies:

```
d r,r
```

Example: Displaying active address spaces

Use the display command to display active address spaces. For example, use the display command to determine if the location service daemon is up:

d a,l	Displays a list of all address spaces.
d a,a	Displays a list of all active address spaces.
d a, <i>address-space-name</i>	Displays only the address space in which you are interested. This command is recommended over the first two because it does not yield such a lengthy list on a production system. Of course, you need to know the name of the address space for which you are looking. Example: d a,bboasr1
d a,bbo*	Displays a list of all active address spaces that start with BBO.

Example: Displaying the status of address spaces registered for automatic restart management

Perform the following steps to use automatic restart management (ARM) to display the status of ARM registered address spaces (including the address spaces of server instances) in the WebSphere for z/OS environment:

1. Initialize all servers.
2. Issue one or both of the following commands:
 - To display all registered address spaces (including the address spaces of server instances), issue the command:
`d xcf,armstatus,detail`
 - To display the status of a particular server instance, use the display command and identify the job name. For example, to display the status of the Daemon server instance (job BBODMN), issue:
`d xcf,armstatus,jobname=bbodmn,detail`

Example: Displaying units of work (transactions) for the Information Management System

You can display units of work (transactions) for Information Management System (IMS):

1. To display the status of a specific transaction, issue the command:
`/dis tran trans-name`
2. To display the status of a specific program, issue the command:
`/dis prog program-name`
3. To display the number of Message Processing Regions (MPRs) that are currently active, issue the command:
`/display active region`

For more information about IMS commands, see the *IMS/ESA Summary of Operator Commands* manual on the IBM Publications Center Web site.

Chapter 11. Welcome to 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. 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.

See **Reference > Javadoc** in the information center navigation for the Javadoc of the JMX APIs.

Chapter 12. Deploying and managing using programming

This section describes how to use Java administrative APIs for customizing the WebSphere Application Server administrative system. 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. For more information, view the JMX Javadoc.

1. Decide that you want to write a Java administrative program. Instead of writing a Java administrative program, consider several other options that you can use to manage WebSphere Application Server processes. These options include: wsadmin, the administrative console, or the administrative command line tools. The administrative tools provide most of the functions you might need to manage the product and the applications that run in the WebSphere Application Server. You can use the command line tools from automation scripts to control the servers. Scripts written for the wsadmin scripting tool offer a wide range of possible custom solutions that you can develop quickly. You might have management features for your application or operating environment that are not implemented in the basic WebSphere Application Server administrative system. In this case, you can use the administrative classes and methods to add newly managed objects to the administrative system. You can also incorporate WebSphere Application Server administration into other Java programs by using the Java administrative APIs.
2. Create a custom Java administrative client program using the Java administrative APIs. (Optional)
3. Extend the WebSphere Application Server administrative system with custom MBeans. (Optional)

Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs

This section describes how to develop a Java program using the WebSphere Application Server administrative APIs for accessing the WebSphere administrative system. For more information, view the JMX Javadoc.

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 the administrative client program by compiling it with javac and providing the location of the necessary JAR files in the classpath argument.

For example, if your installation directory is /DeploymentManager a typical command would look like the following example:(split for publication)

```
javac -classpath
/DeploymentManager/lib/admin.jar;/DeploymentManager/lib/wsexception.jar;
/DeploymentManager/lib/jmxc.jar
MyAdminClient.java
```

(split for publication)

3. Run the administrative client program by setting up the run-time environment so that the program can find all of the prerequisites. Many of the batch or script files in the bin directory under the installation root perform a similar function. An example of a batch file that runs an administrative client program named MyAdminClient follows:

```
@echo off

binDir=~dirname "%0"~
. "%binDir/setupCmdLine.sh"

"$JAVA_HOME/bin/java" "$CLIENTSAS" "-Dwas.install.root=$WAS_HOME"
"-Dwas.repository.root=$CONFIG_ROOT" -Dcom.ibm.CORBA.BootstrapHost=$COMPUTERNAME
-classpath "$WAS_CLASSPATH;/DeploymentManager/classes;/DeploymentManager/lib/admin.jar;
/DeploymentManager/lib/wasjmx.jar" MyAdminClient %*
```

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 Javadoc, the JMX Javadoc, or the MBean Javadoc.

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);
}

```

- 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 executing.
node	The name of the node that the MBean is executing.
process	The name of the process that the MBean is executing.

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");

```

- 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 javadoc. 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);
}

```

- 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 Javadoc. 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 5.0 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 Javadoc.

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 MBeanServer 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.

User MBeans that run on both the WebSphere Application Server distributed platforms and the WebSphere Application Server for z/OS platform may require special coding to function properly in the z/OS multiprocess model. On WebSphere Application Server distributed platforms where each application server runs in a single Java™ Virtual Machine (JVM), there is only one MBean server. The MBean server controls all MBeans that are registered within that application server. On the WebSphere Application Server for z/OS platform, there is a control process and a federation of servant processes, each with their own MBean server. The control process has its own MBean proxy server to distribute requests among the servant processes. See the detailed discussion of the JMX MBean multiprocess model request flow.

2. Register the new MBeans. There are several ways to register your MBean with the MBeanServer in a WebSphere Application Server process. The following list describes the available options:
 - 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.
 - 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.
 - 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.
 - 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.

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.

Java 2 security permissions

You must grant Java 2 security permissions to application scoped code for JMX and WebSphere Application Server administrative privileges in order to allow the code to call WebSphere Application Server administrative and JMX methods.

- To invoke JMX class and interface methods, at least one of the following permissions are required:

```
permission com.tivoli.jmx.MBeanServerPermission "MBeanServer.*"  
permission com.tivoli.jmx.MBeanServerPermission "MBeanServerFactory.*"
```

where the individual target names are:

```
MBeanServer.addNotificationListener  
MBeanServer.createMBean  
MBeanServer.deserialize  
MBeanServer.getAttribute  
MBeanServer.getDefaultDomain  
MBeanServer.getMBeanCount  
MBeanServer.getMBeanInfo  
MBeanServer.getObjectInstance  
MBeanServer.instantiate  
MBeanServer.invoke  
MBeanServer.isRegistered  
MBeanServer.queryMBeans  
MBeanServer.queryNames  
MBeanServer.registerMBean  
MBeanServer.removeNotificationListener  
MBeanServer.setAttribute  
MBeanServer.unregisterMBean  
MBeanServerFactory.createMBeanServer  
MBeanServerFactory.newMBeanServer  
MBeanServerFactory.findMBeanServer  
MBeanServerFactory.releaseMBeanServer
```

- For WebSphere Application Server administrative APIs, the permissions are the following:

```
permission com.ibm.websphere.security.WebSphereRuntimePermission "AdminPermission";
```

Java Management Extensions MBean multiprocess model request flow for WebSphere Application Server for z/OS

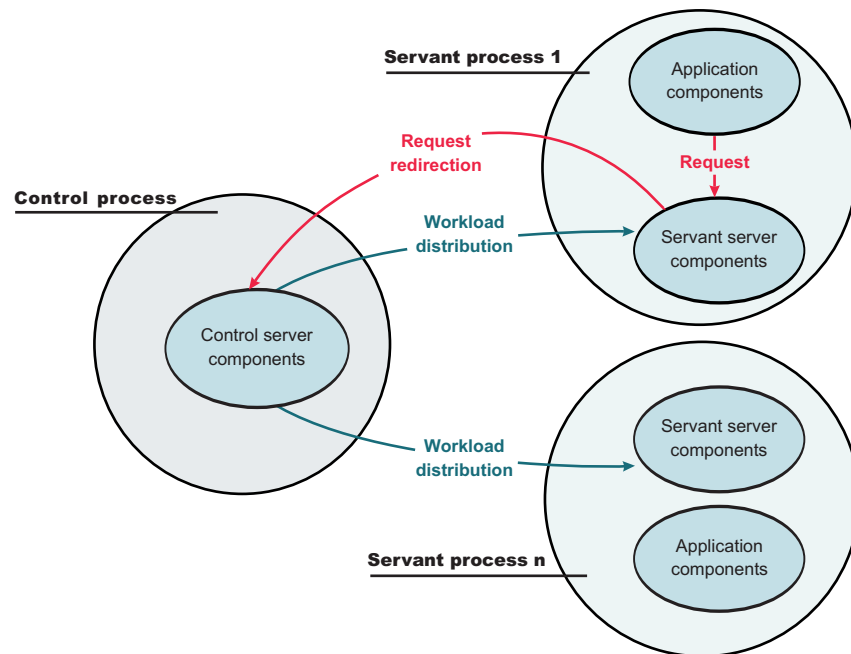
Using the Java Management Extensions (JMX) dynamic proxy capability, applications that depend on JMX operations can exhibit consistent behavior whether the server architecture uses a single process model or a multiprocess model.

All the MBeans that the WebSphere Application Server runtime provides are capable of running under the single process model employed by WebSphere Application Server on distributed platforms, or the multiprocess model employed by WebSphere Application Server for z/OS. User MBean providers might need to modify their MBeans so that they work on both the WebSphere Application Server distributed platforms and the WebSphere Application Server for z/OS. For more information, view the application server API Javadoc.

The simplified dynamic proxy model presented here discusses the two general request flows that exist in the multiprocess model. Operation requests on an MBean can be initiated from one of two places:

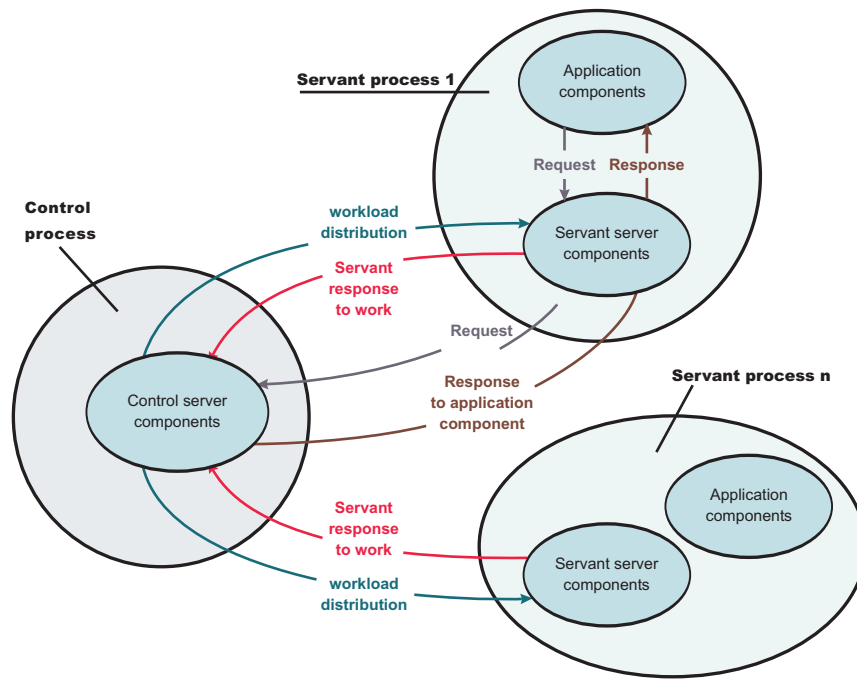
- Within an application component running under the same servant process as the MBean.
- Outside the server through one of the JMX connectors (SOAP, Remote Method Invocation (RMI), HTTP, and so on).

Requests that come from an application component generally follow this flow. The application component sends its request to the servant components. The servant components redirect the request to the control process where the dynamic proxy for the MBean runs. WebSphere Application Server automatically generates the MBean dynamic proxy during runtime. If the request calls a method that the MBean provider defined with a unicast option, the dynamic proxy in the control process randomly dispatches the work, with MVS workload management (WLM), to one servant process. If the request calls a method that the MBean provider defined with a multicast option, the dynamic proxy in the control process, in conjunction with WLM, distributes the work to all the servant processes.



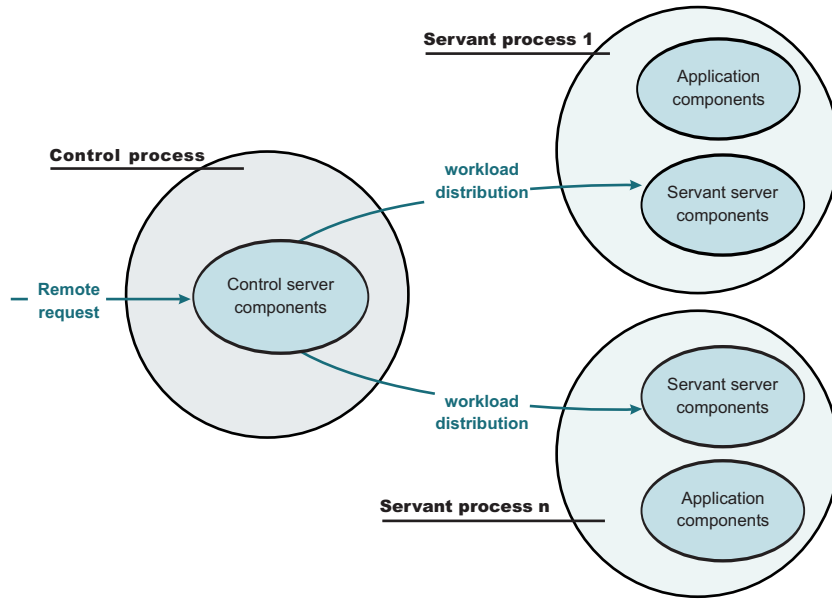
Each servant process that completes the work optionally sends a response back to the control process. If the MBean was defined with a unicast option, and the return type is anything but void, the control process returns the response to the servant that made the request. The servant server components then return the response to the application component. If the MBean was defined with a multicast option, the MBean inside each servant process runs separately and finishes processing the request at different times. After all the requests are processed, you might need a result aggregation and an event aggregation to properly return a result to the

application component.

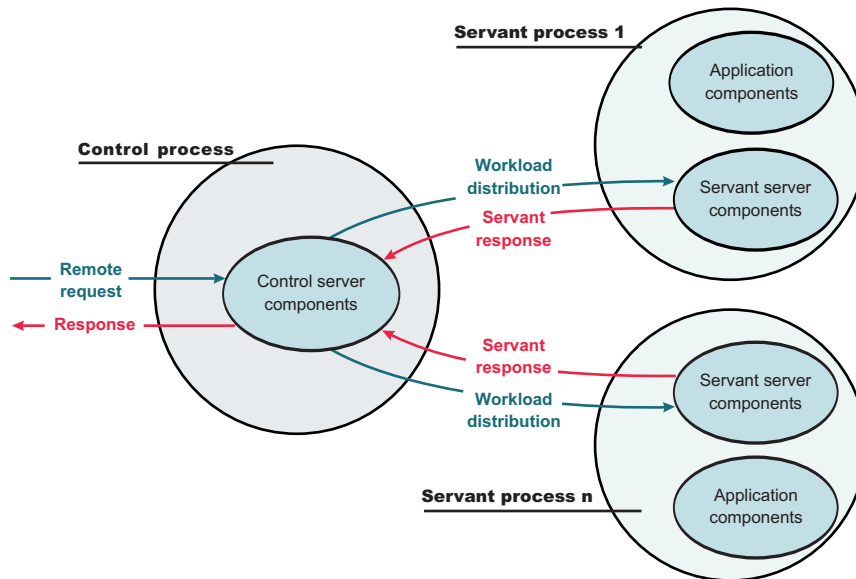


Requests that come from a remote location outside the server generally follow this flow. The remote location sends its request to the control process where the dynamic proxy for the MBean runs. WebSphere Application Server automatically generates the MBean dynamic proxy during run time. The next part of the request process behaves the same as a request that originates from an application component. If the request calls a method that the MBean provider defined with a unicast option the dynamic proxy in the control process randomly dispatches the work, with MVS workload management (WLM), to one servant process. If the request calls a method that the MBean provider defined with a multicast option, the dynamic proxy in the control process, in conjunction with WLM, distributes the

work to all the servant process.



The flow of the response back to the remote location is similar to the response back to the application component. Each servant process that completes the work optionally sends a response back to the control process. If the MBean was defined with a unicast option, and the return type is anything but void, the control process returns the response to the remote location. If the MBean was defined with a multicast option, the MBean that runs inside each servant process runs separately and finishes processing the request at different times. After all the requests are processed, you might need a result aggregation and an event aggregation to properly return a result to the remote location.



Java Management Extensions dynamic proxy concepts

A Java Management Extensions (JMX) dynamic proxy coordinates MBean requests among multiprocess servers. This section discusses the main terms associated with a JMX dynamic proxy.

Control process

Receives requests and distributes them to servant processes so that the application server can do work for the requests.

Servant process

Receives work from the control process and carries out the work.

Unicall option versus the multicall option

Use the unicall option on the proxyInvocationType method when a request invokes an arbitrary servant process or servant processes. Use the multicall option on the proxyInvocationType method when a request goes to multiple servant processes and the servant processes return different results.

The following example shows an MBean descriptor that was developed for a single process model (before) and modified for a multiprocess model (after).

Before

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  version="5.0"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"/>
  <attribute description="The state of the MBean." name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
    targetObjectType="objectReference" type="void">
    <signature>
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
      <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>
</MBean>
```

After

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  version="5.0"
  platform="dynamicproxy"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"/>
  <attribute description="The state of the MBean." name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"/>
  proxyInvokeType="unicall" proxySetterInvokeType="multicall"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
    targetObjectType="objectReference" type="void" proxyInvokeType="multicall">
    <signature>
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>
</MBean>
```

```

        name="mbeanName" type="java.lang.String"/>
        <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
</operation>
</MBean>

```

Make the user MBean run in dynamic proxy mode by specifying `dynamicproxy` on the platform attribute. If no platform attribute exists on the MBean descriptor, the user MBean deployed on WebSphere Application Server for z/OS automatically uses the dynamic proxy mode.

Update the attribute XML tag or the operation XML tag, as shown in the After example, to specify the unicast behavior or the multicast behavior in the multiprocess environment. If no `proxyInvokeType` option or `proxySetterInvokeType` option exists, the behavior defaults to unicast. In the After example, the `getMBeanName` method and the `getState` method run with unicast behavior. The `setState` method and the `initializeState` method run with multicast behavior.

Single process model

The single process application server has one server run time. The MBean generally acts on one instance of each major run time component: one Enterprise Java Beans (EJB) container, one Web container, one Java 2, platform Enterprise Edition (J2EE) connection manager, and so on. This model assumes that each MBean invocation on the server runs in the same process and the same Java Virtual Machine (JVM).

Multiprocess model

The multiprocess model asserts that a single server instance is a federation of Java virtual machines (JVMs), that run in a separate operating process. The control process is responsible for such server functions as communication endpoints, authorization, resource recovery, and workload management. All other JVMs are worker JVMs, in which application requests run. These JVMs take direction from, and interact only with the control process.

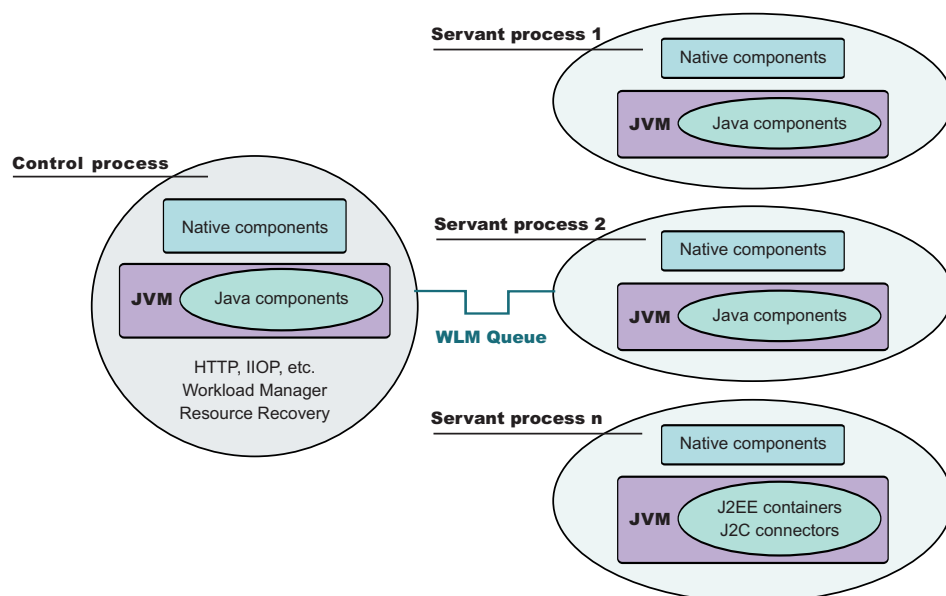
All inbound and outbound requests go through the control process. Client requests arrive at the control process. The control process, with assistance from the MVS workload manager (WLM), dispatches the work to the servant processes.

The number of servant processes is managed by WLM and varies based on demand. The demand is measured against installation-specific performance goals, expressed as WLM policy. Each servant process is identical and hosts the necessary application server components to enable the Java 2, platform Enterprise Edition (J2EE) application programming model. The servant processes rely on the control process for numerous services, such as communication, security, and transaction control.

The multiprocess model imposes additional demands on the Java Management Extension (JMX) infrastructure over the single process model. Administrative requests to a multiprocess server often require coordination among the processes that comprise the application server. The JMX infrastructure includes

additional facilities to enable this coordination.

WebSphere Application Server for z/OS



State object support for dynamic proxy MBean

com.ibm.websphere.management.dynamicproxy.StateObject class: The MBean provider extends the StateObject abstract class. Specify the subclass of the StateObject class so that the JMX run time can instantiate it before the dynamic proxy MBean completes its initialization. The JMX run time attaches StateObject class to the dynamic proxy Invocation Handler interface to keep track of the current state of the dynamic proxy before and after the MBean method runs. The JMX run time also attaches the StateObject class to the Result Aggregation interface class as well as the Event Handler interface class to support appropriate aggregation application.

Result aggregation handler support interface

com.ibm.websphere.management.dynamicproxy.AggregationHandler class: The result aggregation handler support interface defines the method that an MBean provider uses to handle result aggregation in a dynamic proxy-enabled WebSphere Application Server for z/OS MBean. Specify the aggregationHandlerClass attribute on the MBeanDescriptor MBean XML tag. Implement the interface for MBean methods that use the multicall proxyInvokeType option and that return a value. The interface determines the method for which this aggregation is processed. It then properly aggregates all servant MBean results that the servant processes pass back to the control process, and then compiles a single result to return to the caller.

Event aggregation handler support interface

com.ibm.websphere.management.dynamicproxy.EventHandler interface class: The event aggregation handler support interface defines the method that an MBean provider uses to handle event aggregation in a dynamic proxy-enabled WebSphere Application Server for z/OS MBean. Specify the eventHandlerClass attribute on the MBeanDescriptor MBean XML tag. The interface handles all incoming servant MBean events and aggregates them to filter out duplicate events from multiple servant MBeans. It sends one event back to the listener of the dynamic proxy MBean. The interface adjusts the current dynamic proxy MBean state according to the MBean provider requirements.

Invocation handler support interface

com.ibm.websphere.management.dynamicproxy.InvocationHandler class: The invocation handler support interface defines the `preInvoke` and `postInvoke` methods that a WebSphere Application Server for z/OS dynamic proxy MBean implements when it requires state management information. The MBean uses the information to coordinate with the servant MBeans in cases where the multicall invocation type is required. Specify the `invocationHandlerClass` attribute on the `MBeanDescriptor` MBean XML tag. Use the interface for dynamic proxy MBeans that require state management before and after invoking a method that changes its state.

User MBean

The user MBean resides in the servant process and handles requests through its dynamically created proxy MBean, which runs inside the control process. An MBean provider can package handlers with the user MBean so that the provider hooks in his own specialized processing for the following situations:

- Result aggregation
- Event aggregation
- Invocation handling
- State management of objects

Example: The SampleStateMBean MBean

Use this example to guide you in developing user MBeans that work for the WebSphere Application Server on both the distributed platforms and the z/OS platform. The example uses all the special handlers to show its dynamic proxy MBean responsibilities and capabilities. The `SampleStateMBean` example keeps track of its state and generates state change events when it invokes setter methods.

MBeanDescriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "mbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  aggregationHandlerClass="com.ibm.ws390.sample.SampleStateAggregationHandler"
  eventHandlerClass="com.ibm.ws390.sample.SampleStateEventHandler"
  invocationHandlerClass="com.ibm.ws390.sample.SampleStateInvocationHandler"
  stateObjectClass="com.ibm.ws390.sample.SampleState"
  version="5.0"
  platform="dynamicproxy"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"
    proxyInvokeType="unicall"/>
  <attribute description="The state of the MBean." name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"
    proxyInvokeType="multicall" proxySetterInvokeType="multicall"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
    targetObjectType="objectReference" type="void" proxyInvokeType="multicall">
    <signature>
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
      <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>

  <notification name="j2ee.state.starting" severity="6" log="false"
    description="This sample state MBean is in starting state.">
    <notificationType>j2ee.state.starting</notificationType>
```

```

</notification>
<notification name="j2ee.state.running" severity="6" log="false"
    description="This sample state MBean is in running state.">
    <notificationType>j2ee.state.running</notificationType>
</notification>
<notification name="j2ee.state.stopping" severity="6" log="false"
    description="This sample state MBean is in stopping state.">
    <notificationType>j2ee.state.stopping</notificationType>
</notification>
<notification name="j2ee.state.stopped" severity="6" log="false"
    description="This sample state MBean is in stopped state.">
    <notificationType>j2ee.state.stopped</notificationType>
</notification>
</MBean>

```

SampleState implementation

```

package com.ibm.ws390.sample;

import com.ibm.ejs.ras.Tr;
import com.ibm.ejs.ras.TraceComponent;
import java.io.Serializable;
import com.ibm.websphere.management.dynamicproxy.StateObject;

public class SampleState extends StateObject {

    private static TraceComponent tc =
        Tr.register(SampleState.class,"SampleState",null);

    // Package protected STATE constants.
    static final String STATE_STARTING = "j2ee.state.starting";
    static final String STATE_RUNNING = "j2ee.state.running";
    static final String STATE_STOPPING = "j2ee.state.stopping";
    static final String STATE_STOPPED = "j2ee.state.stopped";

    // Dynamicproxy State is initialized with STOPPED state.
    private String state = STATE_STOPPED;

    public SampleState() {
        if (tc.isEntryEnabled()) Tr.entry(tc,"<init>");

        // State is initialized during "state" initialization above,
        // but can also be initialized here in the constructor as well.
        /*
        state = "WebSphere for z/OS App Server ready for e-business";
        */

        if (tc.isEntryEnabled()) Tr.exit(tc,"<init>");
    }

    public synchronized String getState() {
        if (tc.isEntryEnabled()) Tr.entry(tc,"getState");
        if (tc.isEntryEnabled()) Tr.exit(tc,"getState",state);
        return state;
    }

    public synchronized void setState(String state) {
        if (tc.isEntryEnabled()) Tr.entry(tc,"setState",state);
        this.state = state;
        if (tc.isEntryEnabled()) Tr.exit(tc,"setState");
    }

    public synchronized String getStateObjectInfo() {
        return state;
    }
}

```

SampleStateAggregationHandler implementation

```
package com.ibm.ws390.sample;

import com.ibm.websphere.management.dynamicproxy.AggregationHandler;
import com.ibm.websphere.management.dynamicproxy.StateObject;

import com.ibm.ejs.ras.Tr;
import com.ibm.ejs.ras.TraceComponent;

public class SampleStateAggregationHandler implements AggregationHandler {

    private static TraceComponent tc =
        Tr.register(SampleStateAggregationHandler.class,"SampleState",null);

    /**
     * Return an aggregated result from a multicall Mbean operation which
     * compiles through all servant MBeans' results and returns a respective
     * single return value for an invoked method.
     *
     * @param methodName      MBean method name
     * @param params          MBean method parameters
     * @param signatures      MBean method signatures
     * @param servantMBeanResults Result of each servant MBean instances
     *                        invoked by the dynamicproxy multicast
     *                        invocation.
     *
     * Note: this value can be "null" OR can be
     * an array of "null"s in case return value
     * of the method is "void." Implementation
     * of this method MUST handle this case to
     * avoid a <code>NullPointerException</code>.
     *
     * @param stateObject
     * MBean provider provided <code>StateObject</code> used by
     * dynamicproxy MBean in CR to manage its state. Note: this object
     * MAY BE null if "stateObjectClass" was not specified OR internal
     * error occurred during initialization of this dynamicproxy MBean.
     * Implementation MUST properly handle "null" input.
     *
     * @return aggregated result as defined by MBean xml for specified
     * MBean operation.
     */
    public Object aggregateResults(String methodName,
                                  Object[] params,
                                  String[] signatures,
                                  Object[] servantMBeanResults,
                                  StateObject stateObject) {

        if (tc.isEntryEnabled()) Tr.entry(tc,"aggregateResults",methodName);

        // As you can see from the MBeanDescriptor of SampleStateMBean,
        // it declares the following four methods:
        // 1. String getMBeanName()          [proxyInvokeType == unicast]
        // 2. String getState()              [proxyInvokeType == multical]
        // 3. void setState(String)          [proxyInvokeType == multical]
        // 4. void initializeState()         [proxyInvokeType == multical]
        //
        // Looking at the above methods, only method that requires aggregation
        // is #2 getState method which is a multical MBean operation AND
        // it returns a value that can be aggregated.
        //
        // In this example, we simply take each servants' getState MBean
        // request result and concatenate them into one long String that
        // displays each servants' state.
        if (methodName.equals("getState")) {
            StringBuffer stateBuf = new StringBuffer();

            for (int i=0; i<servantMBeanResults.length; i++) {
                stateBuf.append("SERVANT #" + i + " state ==|" +

```



```

        servantMBeanResults[i] + "|== ");
    }
    return stateBuf.toString();
}
// If we also had an example method which returns say an int,
// getNumberOfMBeans(), it can take the similar approach
// and to add each servants' getNumberOfMBeans() result together here.
/* example added for non-existent method: int getNumberOfMBeans()
else if (methodName.equals("getNumberOfMBeans")) {
    int aggregatedResult = 0;

    for (int i=0; i<servantMBeanResults.length; i++) {
        aggregatedResult += (int) servantMBeanResults[i];
    }
    return aggregatedResult;
}
*/
return methodName + " is NOT handled by " + getClass().getName() + "!";
}
}

```

Chapter 13. Welcome to 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 JMX specification. All operations on managed resources in the product go through JMX functions. This means a more standard framework underlying your administrative operations, as well as the ability to tap into the systems management infrastructure programmatically.

Chapter 14. Working with server configuration files

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.

1. Edit configuration files. The master repository is comprised of .xml configuration files. You can edit configuration files using the administrative console, scripting, wsadmin commands, programming, or by editing a configuration file directly. The configuration files are in ASCII format. You cannot edit them in the Hierarchical File System (HFS). Instead, transfer the files to your workstation, edit them, and then transfer them back to the HFS.
2. Save changes made to configuration files. Using the console, you can save changes as follows:
 - a. Click **Save** on the taskbar of the administrative console.
 - b. Put a checkmark in the **Synchronize changes with Nodes** check box.
 - c. On the Save page, click **Save**.
3. Handle temporary configuration files resulting from a session timing out.
4. Change the location of temporary configuration files.
5. Change the location of backed-up configuration files.
6. Change the location of temporary workspace files.
7. Back up and restore configurations.

Configuration documents

WebSphere Application Server stores configuration data for servers in several documents in a cascading hierarchy of directories. The configuration documents describe the available application servers, their configurations, and their contents. Most configuration documents have XML content.

Hierarchy of directories of documents

The cascading hierarchy of directories and the documents' structure support multi-node replication to synchronize the activities of all servers in a cell. In a Network Deployment environment, changes made to configuration documents in the cell repository, are automatically replicated to the same configuration documents that are stored on nodes throughout the cell.

At the top of the hierarchy is the **cells** directory. It holds a subdirectory for each cell. The names of the cell subdirectories match the names of the cells. For example, a cell named *cell1* has its configuration documents in the subdirectory *cell1*.

On the Network Deployment node, the subdirectories under the cell contain the entire set of documents for every node and server throughout the cell. On other nodes, the set of documents is limited to what applies to that specific node. If a configuration document only applies to *node1*, then that document exists in the configuration on *node1* and in the Network Deployment configuration, but not on any other node in the cell.

Each cell subdirectory has the following files and subdirectories:

- The `cell.xml` file, which provides configuration data for the cell
- Files such as `security.xml`, `virtualhosts.xml`, `resources.xml`, and `variables.xml`, which provide configuration data that applies across every node in the cell
- The **clusters** subdirectory, which holds a subdirectory for each cluster defined in the cell. The names of the subdirectories under clusters match the names of the clusters.

Each cluster subdirectory holds a `cluster.xml` file, which provides configuration data specifically for that cluster.

- The **nodes** subdirectory, which holds a subdirectory for each node in the cell. The names of the nodes subdirectories match the names of the nodes.

Each node subdirectory holds files such as `variables.xml` and `resources.xml`, which provide configuration data that applies across the node. Note that these files have the same name as those in the containing cell's directory. The configurations specified in these node documents override the configurations specified in cell documents having the same name. For example, if a particular variable is in both cell- and node-level `variables.xml` files, all servers on the node use the variable definition in the node document and ignore the definition in the cell document.

Each node subdirectory holds a subdirectory for each server defined on the node. The names of the subdirectories match the names of the servers. Each server subdirectory holds a `server.xml` file, which provides configuration data specific to that server. Server subdirectories might hold files such as `security.xml`, `resources.xml` and `variables.xml`, which provide configuration data that applies only to the server. The configurations specified in these server documents override the configurations specified in containing cell and node documents having the same name.

- The **applications** subdirectory, which holds a subdirectory for each application deployed in the cell. The names of the applications subdirectories match the names of the deployed applications.

Each deployed application subdirectory holds a `deployment.xml` file that contains configuration data on the application deployment. Each subdirectory also holds a **META-INF** subdirectory that holds a J2EE application deployment descriptor file as well as IBM deployment extensions files and bindings files. Deployed application subdirectories also hold subdirectories for all `.war` and entity bean `.jar` files in the application. Binary files such as `.jar` files are also part of the configuration structure.

An example file structure is as follows:

```
cells
  cell1
    cell.xml resources.xml virtualhosts.xml variables.xml security.xml
    nodes
      nodeX
        node.xml variables.xml resources.xml serverindex.xml
        serverA
          server.xml variables.xml
        nodeAgent
          server.xml variables.xml
      nodeY
        node.xml variables.xml resources.xml serverindex.xml
    applications
      sampleApp1
        deployment.xml
        META-INF
          application.xml ibm-application-ext.xml ibm-application-bnd.xml
      sampleApp2
```

```

deployment.xml
META-INF
application.xml ibm-application-ext.xml ibm-application-bnd.xml

```

Changing configuration documents

You can use one of the administrative tools (console, wsadmin, Java APIs) to modify configuration documents or edit them directly. It is preferable to use the administrative console because it validates changes made to configurations. *“Configuration document descriptions”* states whether you can edit a document using the administrative tools or must edit it directly.

Configuration document descriptions

Most configuration documents have XML content. The table below describes the documents and states whether you can edit them using an administrative tool or must edit them directly.

If possible, edit a configuration document using the administrative console because it validates any changes that you make to configurations. You can also use one of the other administrative tools (wsadmin or Java APIs) to modify configuration documents. Using the administrative console or wsadmin scripting to update configurations is less error prone and likely quicker and easier than other methods.

However, you cannot edit some files using the administrative tools. Configuration files that you must edit manually have an X in the **Manual editing required** column in the table below. Since the configuration files are in ASCII format, you cannot edit them in the Hierarchical File System (HFS). Instead, transfer the files to your workstation, edit them, and then transfer them back to the HFS.

Document descriptions

(Locations split for publishing)

Configuration file	Locations	Purpose	Manual editing required
admin-authz.xml	config/cells/ <i>cell_name/</i>	Define a role for administrative operation authorization.	X
app.policy	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i>	Define security permissions for application code.	X
cell.xml	config/cells/ <i>cell_name/</i>	Identify a cell.	
deployment.xml	config/cells/ <i>cell_name/</i> applications/ <i>application_name/</i>	Configure application deployment settings such as target servers and application-specific server configuration.	
filter.policy	config/cells/ <i>cell_name/</i>	Specify security permissions to be filtered out of other policy files.	X

integral-jms-authorizations.xml	config/cells/ <i>cell_name/</i>	Provide security configuration data for the integrated messaging system.	X
library.policy	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i>	Define security permissions for shared library code.	X
namestore.xml	config/cells/ <i>cell_name/</i>	Provide persistent name binding data.	X
naming-authz.xml	config/cells/ <i>cell_name/</i>	Define roles for a naming operation authorization.	X
node.xml	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i>	Identify a node.	
resources.xml	config/cells/ <i>cell_name/</i> config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i> config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i> servers/ <i>server_name/</i>	Define operating environment resources, including JDBC, JMS, JavaMail, URL, JCA resource providers and factories.	
security.xml	config/cells/ <i>cell_name/</i>	Configure security, including all user ID and password data.	
server.xml	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i> servers/ <i>server_name/</i>	Identify a server and its components.	
serverindex.xml	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i>	Specify communication ports used on a specific node.	
spi.policy	config/cells/ <i>cell_name/</i> nodes/ <i>node_name/</i>	Define security permissions for service provider libraries such as resource providers.	X

variables.xml	config/cells/ <i>cell_name</i> / config/cells/ <i>cell_name</i> / nodes/ <i>node_name</i> / config/cells/ <i>cell_name</i> / nodes/ <i>node_name</i> / servers/ <i>server_name</i> /	Configure variables used to parameterize any part of the configuration settings.	
virtualhosts.xml	config/cells/ <i>cell_name</i> /	Configure a virtual host and its MIME types.	

Object names

When you create a new object using the administrative console or a wsadmin command, you often must specify a string for a name attribute. Most characters are allowed in the name string. However, the name string cannot contain the following characters:

/	forward slash
\	backslash
*	asterisk
,	comma
:	colon
;	semi-colon
=	equal sign
+	plus sign
?	question mark
	vertical bar
<	left angle bracket
>	right angle bracket
&	ampersand (and sign)
%	percent sign
'	single quote mark
"	double quote mark
]]>	
.	period (not valid if first character; valid if a later character)

Configuration repositories

A configuration repository stores configuration data. By default, configuration repositories reside in the *config* subdirectory of the product installation root directory.

A cell-level repository stores configuration data for the entire cell and is managed by a file repository service that runs in the deployment manager. The deployment manager and each node have their own repositories. A node-level repository stores configuration data needed by processes on that node and is accessed by the node agent and application servers on that node.

When you change a WebSphere Application Server configuration by creating an application server, installing an application, changing a variable definition or the like, and then save the changes, the cell-level repository is updated. The file synchronization service distributes the changes to the appropriate nodes.

Handling temporary configuration files resulting from session timeout

If the console is not used for 15 minutes or more, the session times out. The same thing happens if you close the browser window without saving the configuration file. Changes to the file are saved to a temporary file when the session times out, after 15 minutes.

When a session times out, the configuration file in use is saved under the `userid/timeout` directory under the ServletContext's temp area. This is value of the `javax.servlet.context.tempdir` attribute of the ServletContext. By default, it is:
`install_root/temp/hostname/Administration/admin/admin.war`

You can change the temp area by specifying it as a value for the `tempDir` init-param of the action servlet in the deployment descriptor (`web.xml`) of the administrative application.

The next time you log on to the console, you are prompted to load the saved configuration file. If you decide to load the saved file:

1. If a file with the same name exists in the `install_root/config` directory, that file is moved to the `userid/backup` directory in the temp area.
2. The saved file is moved to the `install_root/config` directory.
3. The file is then loaded.

If you decide not to load the saved file, it is deleted from the `userid/timeout` directory in the temp area.

The configuration file is also saved automatically when the same user ID logs into the non-secured console again, effectively starting a different session. This process is equivalent to forcing the existing user ID out of session, similar to a session timing out.

Changing the location of temporary configuration files

The configuration repository uses copies of configuration files and temporary files while processing repository requests. It also uses a back-up directory while managing the configuration. You can change the default locations of these files from the configuration directory to a directory of your choice using system variables or the administrative console.

The default location for the configuration temporary directory is `CONFIG_ROOT/temp`. Change the location by doing either of the following:

- Set the system variable `was.repository.temp` to the location you want for the repository temporary directory. Set the system variable when launching a Java process using the `-D` option. For example, to set the default location of the repository temporary directory, use the following option:

```
-Dwas.repository.temp=%CONFIG_ROOT%/temp
```

- Use the administrative console to change the location of the temporary repository file location for each server configuration. For example, on the Network Deployment product, to change the setting for a deployment manager, do the following:
 1. Click **System Administration > Deployment Manager** in the navigation tree of the administrative console. Then, click **Administration Services, Repository Service, and Custom Properties**.
 2. On the Properties page, click **New**.
 3. On the settings page for a property, define a property for the temporary file location. The key for this property is `was.repository.temp`. The value can include WebSphere Application Server variables such as `${WAS_TEMP_DIR}/config`. Then, click **OK**.

The system property set using the first option takes precedence over the configuration property set using the second option.

Changing the location of backed-up configuration files

During administrative processes like adding a node to a cell or updating a file, configuration files are backed up to a back-up location. The default location for the back-up configuration directory is `CONFIG_ROOT/backup`. Change the location by doing either of the following:

- Set the system variable `was.repository.backup` to the location you want as the repository back-up directory. Set the system variable when launching a Java process using the `-D` option. For example, to set the default location of the repository back-up directory, use the following option:


```
-Dwas.repository.backup=%CONFIG_ROOT%/backup
```
- Use the administrative console to change the location of the repository back-up directory for each server configuration. For example, on the Network Deployment product, do the following to change the setting for a deployment manager:
 1. Click **System Administration > Deployment Manager** in the navigation tree of the administrative console. Then, click **Administration Services, Repository Service, and Custom Properties**.
 2. On the Properties page, click **New**.
 3. On the settings page for a property, define a property for the back-up file location. The key for this property is `was.repository.backup`. The value can include WebSphere Application Server variables such as `${WAS_TEMP_DIR}/backup`. Then, click **OK**.

The system property set using the first option takes precedence over the configuration property set using the second option.

Changing the location of temporary workspace files

The administrative console workspace allows client applications to navigate the configuration. Each workspace has its own repository location defined either in the system property or the property passed to a workspace manager when creating the workspace: `workspace.user.root` or `workspace.root`, which is calculated as `%workspace.root%/user_ID/workspace/wstemp`.

The default workspace root is calculated based on the user installation root: `%user.install.root%/wstemp`. You can change the default location of temporary workspace files by doing either of the following:

- Change the setting for the system variable *workspace.user.root* or *workspace.root* so its value is no longer set to the default location. Set the system variable in the servant process when launching a Java process using the -D option. For example, to set the default location the full path of the root of all users' directories, use the following option:
`-Dworkspace.user.root=full_path_for_root_of_all_user_directories`
- Change the setting at the user level with the programming APIs. When you create the workspace, set the `WorkSpaceManager.WORKSPACE_USER_ROOT` property on the API.

Backing up and restoring administrative configurations

WebSphere Application Server represents its administrative configurations as XML files. You should back up configuration files on a regular basis.

1. Synchronize administrative configuration files.
 - a. Click **System Administration > Nodes** in the console navigation tree to access the Nodes page.
 - b. Click **Full Resynchronize**. The resynchronize operation resolves conflicts among configuration files and can take several minutes to run.
2. Run the `backupConfig` command to back up configuration files.
3. Run the `restoreConfig` command to restore configuration files. Specify backup files that do not contain invalid or inconsistent configurations.

Server configuration files: Resources for learning

Use the following links to find relevant supplemental information about administering WebSphere Application Server server configuration files. 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

- Listing of all IBM WebSphere Application Server Redbooks
- System Administration for WebSphere Application Server V5 -- Part 1: Overview of V5 Administration

Backing up the WebSphere for z/OS system

Use the following guidelines to back up parts of your WebSphere for z/OS system.

1. **Back up the HFS that contains your WebSphere for z/OS configuration** (e.g. `WebSphere/V5R0M0/AppServer`).
2. Back up the RMDATA log for Resource Recovery Service (RRS). Otherwise, a failure could force you to do a cold start of RRS.
3. Set the ARCHIVE log retention period to one day.
4. Incorporate the following items in your normal backup procedures:

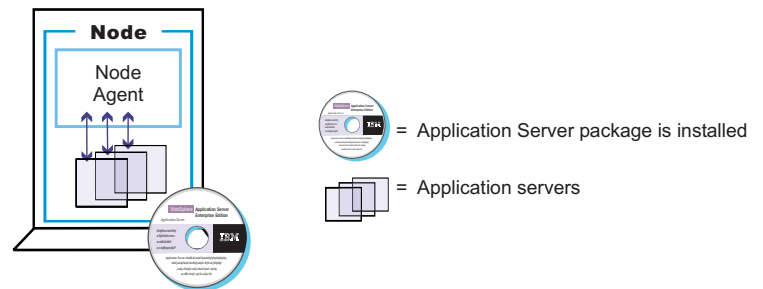
- WebSphere for z/OS procedure libraries.
 - WebSphere for z/OS load libraries.
 - The directory where WebSphere for z/OS run-time information is written. The default is */WebSphere/V5R0M0*.
5. Back up your own application executables, databases, and bindings.
 6. If you wish to back up a single server, you can use the export/import function in the Administrative Console. For details on how to do this, see the assembling applications information.

Chapter 15. Welcome to 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.

A base WebSphere Application Server (single server) installation includes only the Application Server process. A single server installation can host one or more sets of managed servers, known as nodes. A managed server is a single WebSphere Application Server JVM instance, running in its own process. A node cannot span multiple machines, but a machine can have multiple nodes, each with multiple managed servers. There is no node agent or network deployment manager involved in this configuration. No coordination between application server processes is supported in the single server environment. Administration is limited

IBM WebSphere Application Server Package



to a single process at a time.

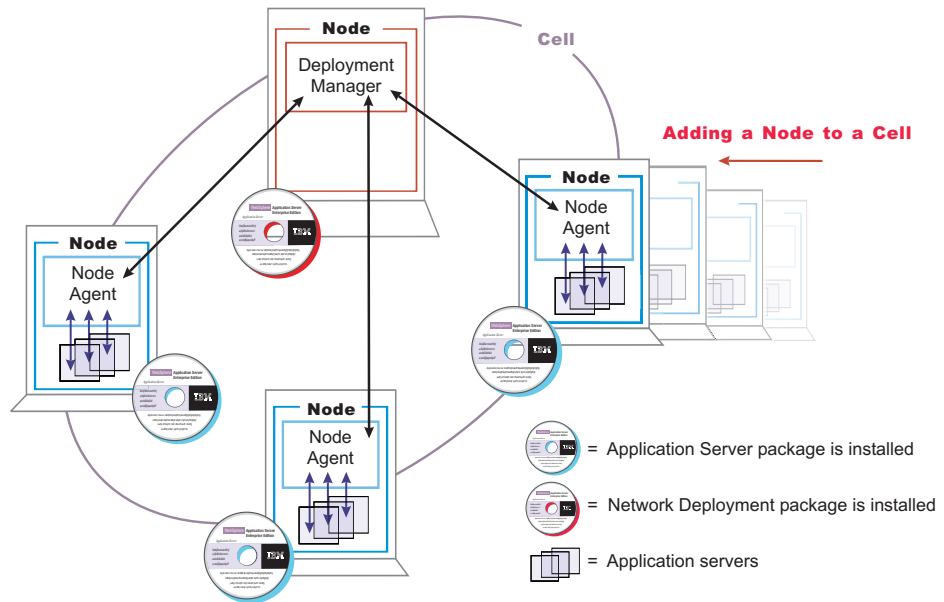
A Network Deployment installation can support a network of computer systems that are configured to run collaborating instances of a single server installation. Each computer having a single server installation is known as a node. The Network Deployment product provides centralized administration and workload management for a set of nodes, known as a cell. A cell has a master administrative repository that stores all of the cell's configuration data.

One computer is designated as the central deployment manager machine. The central deployment manager program (dmgr) oversees the cell.

In the Network Deployment product, each node has a node agent that serves as an intermediary between the application servers on the node and the deployment manager that oversees the entire cell.

Administrative logic running in a node agent keeps the configuration data of a node synchronized with the configuration data of all nodes within a cell.

IBM WebSphere Application Server Network Deployment Package



You establish a multiple machine environment (a cell) through a series of installation and configuration steps.

Chapter 16. Managing administrative agents

After you set up the Network Deployment environment, you probably will not need to change the configuration of its components much, if at all. You might want to manage configuration data stored in files by configuring a file synchronization service (part of managing nodes in step 5). However, you will mainly need to monitor and control incorporated nodes, and the resources on those nodes, using the administrative console or other administrative tools.

1. Use the settings page for an administrative service to configure administrative services.
2. Use the settings page for a repository service to configure administrative repositories.
3. Configure cells.
4. Configure deployment managers.
5. Manage nodes.
6. Manage node agents.
7. Configure remote file services.
8. Configuring location service daemons.

Cells

Cells are arbitrary, logical groupings of one or more nodes in a WebSphere Application Server distributed network.

A cell is a configuration concept, a way for administrators to logically associate nodes with one another. Administrators define the nodes that make up a cell, according to the specific criteria that make sense in their organizational environments.

Administrative configuration data is stored in XML files. A cell retains master configuration files for each server in every node in the cell. Each node and server also have their own local configuration files. Changes to a local node or to a server configuration file are temporary, if the server belongs to the cell. While in effect, local changes override cell configurations. Changes to server and node configuration files at the cell level are permanent. Synchronization occurs at designated events, such as when a server starts.

Configuring 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. To view information about and manage a cell, use the settings page for a cell.

1. Access the settings page for a cell. Click **System Administration > Cell** from the navigation tree of the administrative console.
2. If the protocol that the cell uses to retrieve information from a network is not appropriate for your system, select the appropriate protocol. By default, a cell uses Transmission Control Protocol (TCP). If you want the cell to use User Datagram Protocol, select **UDP** from the drop-down list for **Cell Discovery**

Protocol on the settings page for the cell. It is unlikely that you will need to change the cell's protocol configuration from TCP.

3. Specify an end point to hold the discovery address. An end point designates a host name and port. You can specify any string; for example, `my_cell_endpoint`. It is unlikely that you will need to change the cell's end point configuration.
4. Click **Properties** and define any name-value pairs needed by your deployment manager.
5. When you installed the WebSphere Application Server Network Deployment product, a node was added to the cell. You can add additional nodes on the Node page. Click **Nodes** to access the Node page, which you use to manage nodes.

Cell settings

Use this page to set the discovery protocol and address end point for an existing cell. A cell is a configuration concept, a way for an administrator to logically associate nodes according to whatever criteria make sense in the administrator's organizational environment.

To view this administrative console page, click **System Administration > Cell**.

Name

Specifies the name of the existing cell.

Data type String

Short Name

Specifies the short name of the cell. The name is 1-8 characters, alpha-numeric or national language. It cannot start with a numeric.

The short name property is read only. It was defined during installation and customization.

Cell Discovery Protocol

Specifies the protocol that the cell follows to retrieve information from a network.

Select one of these protocol options:

UDP User Datagram Protocol (UDP)

TCP Transmission Control Protocol (TCP)

Data type String

Default TCP

Discovery Address Endpoint Name

Specifies the name of the end point that contains the discovery address.

Data type String

Deployment managers

Deployment managers are administrative agents that provide a centralized management view for all nodes in a cell, as well as management of clusters and workload balancing of application servers across one or several nodes in some editions. WebSphere Application Server for z/OS uses Workload Management (WLM) as the primary vehicle for workload balancing.

A deployment manager hosts the administrative console. A deployment manager provides a single, central point of administrative control for all elements of the entire WebSphere Application Server distributed cell.

Configuring deployment managers

When you installed the WebSphere Application Server Network Deployment product, a deployment manager was created. A deployment manager provides a single, central point of administrative control for all elements in a WebSphere Application Server distributed cell. You probably will not need to reconfigure the deployment manager. To view information about and manage a deployment manager, use the settings page for a deployment manager.

1. Access the settings page for a deployment manager. Click **System Administration > Deployment Manager** from the navigation tree of the administrative console.
2. Configure the deployment manager as desired by clicking on a property such as **Custom Services** and specifying settings on the resulting pages. A deployment manager is like an application server and you can configure a deployment manager in a manner similar to the way you configure an application server.

Deployment manager settings

Use this page to name a deployment manager, to stop its running, and to link to other pages which you can use to define additional properties for the deployment manager. A deployment manager provides a single, central point of administrative control for all elements of the entire WebSphere Application Server distributed cell.

To view this administrative console page, click **System Administration > Deployment Manager**.

Name

Specifies a logical name for the deployment manager. The name must be unique within the cell.

Data type

String

Short name

Specifies the short name of the deployment manager server. The server short name must be unique within a cell. The short name identifies the server to the native facilities of the operating system, such as Workload Manager (WLM), Automatic Restart Manager, SAF (for example, RACF), started task control, and others.

The name is 1-8 characters, alpha numeric or national language. It cannot start with a numeric.

The system assigns a cell-unique, default short name.

Unique Id

Specifies the unique ID of this deployment manager server.

The unique ID property is read only. The system automatically generates the value.

Process ID

Specifies a string identifying the process.

Data type	String
Default	None

Cell Name

Specifies the name of the cell for the deployment manager. The default is the name of the host computer on which the deployment manager is installed with *Network* appended.

Data type	String
Default	<i>host_name</i> Network

Node Name

Specifies the name of the node for the deployment manager. The default is the name of the host computer on which the deployment manager is installed with *Manager* appended.

Data type	String
Default	<i>host_name</i> Manager

State

Indicates the state of the deployment manager. The state is *Started* when the deployment manager is running and *Stopped* when it is not running.

Data type	String
Default	Started

Node

A node is a logical grouping of managed servers.

A node usually corresponds to a physical computer system with a distinct IP host address. Node names usually are identical to the host name for the computer.

A node agent manages all WebSphere Application Server servers on a node. The node agent represents the node in the management cell.

Managing nodes

A node is a grouping of managed servers. To view information about and manage nodes, use the Nodes page. To access the Nodes page, click **System Administration > Nodes** in the console navigation tree.

1. Add a node.
 - a. Ensure that an application server is running on the remote host for the node that you are adding. Also ensure that the application server has a SOAP connector on the port for the host.

- b. Go to the Nodes page and click **Add Node**. On the Add Node page, specify a host name and SOAP connector port for the deployment manager, then click **OK**.

The node is added to the WebSphere Application Server environment and the name of the node appears in the collection on the Nodes page.

2. If the discovery protocol that a node uses is not appropriate for the node, select the appropriate protocol. On the Nodes page, click on the node to access the settings for the node. Select a value for **Discovery Protocol**. By default, a node uses Transmission Control Protocol (TCP). You will likely not need to change a node's protocol configuration from TCP. However, if you do need to change the discovery protocol value, here are some guidelines:
 - For a managed process, use **multicast**. A ManagedProcess supports multicast only because multicasting allows all application servers in one node to listen to one port instead of to one port for each server. A benefit of using multicast is that you do not have to configure the discovery port for each application server or prevent conflicts in ports. A drawback of using multicast is that you must ensure that your machine is connected to the network when application servers (not including the node agent) launch because a multicast address is shared by the network and not by the local machine. If your machine is not connected to the network when application servers launch, the multicast address will not be shared with the application servers.
 - For a node agent or deployment manager, use **TCP** or **UDP**. Do not use **multicast**.
3. Define a custom property for a node.
 - a. On the Nodes page, click on the node for which you want to define a custom property.
 - b. On the settings for the node, click **Custom Properties**.
 - c. On the Property collection page, click **New**.
 - d. On the settings page for a property instance, specify a name-value pair and a description for the property, then click **OK**.
4. If you added a node or changed a node's configuration, synchronize the node's configuration. On the Node Agents page, ensure that the node agent for the node is running. Then, on the Nodes page, put a checkmark in the check box beside the node whose configuration files you want to synchronize and click **Synchronize** or **Full Resynchronize**.

Clicking either button sends a request to the node agent for that node to perform a configuration synchronization immediately, instead of waiting for the periodic synchronization to occur. This is important if automatic configuration synchronization is disabled, or if the synchronization interval is set to a long time, and a configuration change has been made to the cell repository that needs to be replicated to that node. Settings for automatic synchronization are on the File Synchronization Service page.

Synchronize requests that a node synchronization operation be 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. So it is still possible for the node and cell configuration to be out of synchronization after this operation is performed.

Full Resynchronize clears all synchronization optimization settings and performs configuration synchronization anew, so there will be no mismatch between node and cell configuration after this operation is performed. This operation can take longer than the **Synchronize** operation.

5. Stop servers on a node. On the Nodes page, put a checkmark in the check box beside the node whose servers you want to stop running and click **Stop**.

6. Remove a node. On the Nodes page, put a checkmark in the check box beside the node you want to delete and click **Remove Node**.

Node collection

Use this page to manage nodes in the WebSphere environment. Nodes group managed servers.

To view this administrative console page, click **System Administration > Nodes**.

Name

Specifies a name for a node that is unique within the cell.

A node name is identical to the host name for the computer. That is, a node corresponds to a physical computer system with a distinct IP host address.

Status

Indicates the state of the node.

The state can be **Started**, **Stopped**, **Unavailable**, **Unknown**, **Partial Start**, **Partial Stop**, **Synchronized**, or **Not synchronized**.

Node settings

Use this page to view or change the configuration or topology settings for a node instance.

To view this administrative console page, click **System Administration > Nodes >node_name**.

Name:

Specifies a logical name for the node. The name must be unique within the cell.

A node name usually is identical to the host name for the computer.

Data type String

Short Name:

Specifies the name of a node. The name is 1-8 characters, alpha-numeric or national language. It cannot start with a numeric.

The short name property is read only. It is defined during installation and customization.

Discovery Protocol:

Specifies the protocol that servers use to discover the presence of other servers.

Select from one of these protocol options:

UDP User Datagram Protocol (UDP)

TCP Transmission Control Protocol (TCP)

multicast

IP multicast protocol

Data type String

Default	TCP
Range	Valid values are UDP, TCP, or multicast.

Administration service settings

Use this page to view and change the configuration for an administration service.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers >*server_name* > Administration Services**
- **Servers > JMS Servers >*server_name* > Administration Services**

Standalone

Specifies whether the server process is a participant in a Network Deployment cell or not. If the box is checked (true), the server does not participate in distributed administration. If the box is unchecked (false), the server participates in the Network Deployment system.

The default value for base WebSphere Application Server installations is true. When addNode runs to incorporate the server into a Network Deployment cell, the value switches to false.

Data type	Boolean
Default	true

Preferred Connector

Specifies the preferred JMX Connector type. Available options, such as SOAPConnector or RMICConnector, are defined using the JMX Connectors page.

Data type	String
Default	SOAP

Extension MBean Providers collection

Use this page to view and change the configuration for JMX extension MBean providers.

You can configure JMX extension MBean providers to be used to extend the existing WebSphere managed resources in the core administrative system. Each MBean provider is a library containing an implementation of a JMX MBean and its MBean XML Descriptor file.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers >*server_name* > Administration Services > Extension MBean Providers**
- **Servers > JMS Servers >*server_name* > Administration Services > Extension MBean Providers**

Classpath

The classpath within the provider library where the MBean Descriptor can be located.

Description

An arbitrary descriptive text for the Extension MBean Provider configuration.

Name The name used to identify the Extension MBean provider library.

Extension MBean Provider settings

Use this page to view and change the configuration for a JMX extension MBean provider.

You can configure a library containing an implementation of a JMX MBean, and its MBean XML Descriptor file, to be used to extend the existing WebSphere managed resources in the core administrative system

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers** >*server_name* > **Administration Services > Extension MBean Providers** >*provider_library_name*
- **Servers > JMS Servers** >*server_name* > **Administration Services > Extension MBean Providers** >*provider_library_name*

Classpath:

The classpath within the provider library where the MBean Descriptor can be located. The class loader needs this information to load and parse the Extension MBean XML Descriptor file.

Data type String

Description:

An arbitrary descriptive text for the Extension MBean Provider configuration. Use this field for any text that helps identify or differentiate the provider configuration.

Data type String

Name:

The name used to identify the Extension MBean provider library.

Data type String

Repository service settings

Use this page to view and change the configuration for an administrative service repository.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers** >*server_name* > **Administration Services > Repository Service**
- **Servers > JMS Servers** >*server_name* > **Administration Services > Repository Service**

Audit Enabled

Specifies whether to audit repository updates in the log file. The default is to audit repository updates.

Data type Boolean
Default true

Node agents

Node agents are administrative agents that route administrative requests to servers.

A node agent is a server that runs on every host computer system that participates in the WebSphere Application Server Network Deployment product. It is purely an administrative agent and is not involved in application serving functions. A node agent also hosts other important administrative functions such as file transfer services and configuration synchronization.

Managing 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.

1. View information about a node agent. Use the Node Agents page. Click **System Administration > Node Agents** in the console navigation tree. To view additional information about a particular node agent or to further configure a node agent, click on the node agent's name under **Name**.
2. Stop and then restart all application servers on the node managed by the node agent. On the Node Agents page, place a checkmark in the check box beside the node agent that manages the node whose servers you want to restart, then click **Restart all Servers on Node**. Note that the node agent for the node must be processing in order to restart application servers on the node.
3. Stop the processing of a node agent. On the Node Agents page, place a checkmark in the check box beside the node agent you want to stop processing, then click **Stop**.

Node agent collection

Use this page to view information about node agents. Node agents are administrative agents that monitor application servers on a host system and route administrative requests to servers. A node agent is the running server that represents a node in a Network Deployment environment.

To view this administrative console page, click **System Administration > Node Agents** .

Name

Specifies a logical name for the node agent server.

Node

Specifies a name for the node. The node name is unique within the cell.

A node name usually is identical to the host name for the computer. That is, a node usually corresponds to a physical computer system with a distinct IP host address.

However, the node name is a purely logical name for a group of servers. You can name the node anything you please. The node name does not have to be the host name.

Status

Indicates whether the node agent server is started or stopped.

Note that if the status of servers such application servers is *Unavailable*, the node agent is not running in the servers' node and you must restart the node agent before you can start the servers.

Node agent server settings

Use this page to view information about and configure a node agent. A node agent coordinates administrative requests and event notifications among servers on a machine. A node agent is the running server that represents a node in a Network Deployment environment.

To view this administrative console page, click **System Administrator > Node Agents > *node_agent_name***.

A node agent must be started on each node in order for the deployment manager node to be able to collect and control servers configured on that node. If you use configuration synchronization support, a node agent coordinates with the deployment manager server to synchronize the node's configuration data with the master copy managed by the deployment manager.

Name:

Specifies a logical name for the node agent server.

Data type	String
Default	NodeAgent Server

Short name:

Specifies the short name of the node agent server. The server short name must be unique within a cell. The short name identifies the server to the native facilities of the operating system, such as Workload Manager (WLM), Automatic Restart Manager, SAF (for example, RACF), started task control, and others.

The name is 1-8 characters, alpha-numeric or national language. It cannot start with a numeric.

The system assigns a cell-unique, default short name.

Unique Id:

Specifies the unique ID of this node agent server.

The unique ID property is read only. The system automatically generates the value.

Process ID:

Specifies a string identifying the process.

Data type	String
------------------	--------

Cell Name:

Specifies the name of the cell for the node agent server.

Data type	String
Default	<i>host_name</i> Network

Node Name:

Specifies the name of the node for the node agent server.

Data type	String
------------------	--------

State:

Indicates whether the node agent server is started or stopped.

Data type	String
Default	Started

Remote file services

Configuration documents describe the available application servers, their configurations, and their contents. Two file services manage configuration documents: the file transfer service and the file synchronization service.

The file services do the following:

File transfer service

The file transfer service enables the moving of files between the network manager and the nodes. It uses the HTTP protocol to transfer files. When you enable security in the WebSphere Application Server product, the file transfer service uses certificate-based mutual authentication. You can use the default key files in a test environment. Ensure that you change the default key file to secure your system.

The ports used for file transfer are defined in the Network Deployment server configuration under its WebContainer HTTP transports.

File synchronization service

The file synchronization service ensures that a file set on each node matches that on the deployment manager node. This service promotes consistent configuration data across a cell. You can adjust several configuration settings to control file synchronization on individual nodes and throughout a system.

This service runs in the deployment manager and node agents, and ensures that configuration changes made to the cell repository are propagated to the appropriate node repositories. The cell repository is the master repository, and configuration changes made to node repositories are not propagated up to the cell. During a synchronization operation a node agent checks with the deployment manager to see if any configuration documents that apply to the node have been updated. New or updated documents are copied to the node repository, and deleted documents are removed from the node repository.

The default behavior, which is enabled, is for each node agent to periodically run a synchronization operation. You can configure the

interval between operations or disable the periodic behavior. You can also configure the synchronization service to synchronize a node repository before starting a node on the server.

Configuring 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.

1. Go to the File Synchronization Service page. Click **System Administration > Node Agents** in the console navigation tree. Then, click the node agent for which you want to configure a synchronization server and click **File Synchronization Service**.
2. On the File Synchronization Service page, customize the service that helps make configuration data consistent across a cell by moving updated configuration files from the deployment manager to the node. Change the values for properties on the File Synchronization Service page. The file synchronization service is always started, but you can control how it runs by changing the values.

File transfer service settings

Use this page to configure the service that transfers files from the deployment manager to individual remote nodes.

To view this administrative console page, click **System Administration > Node Agents > *node_agent_name* > File Transfer Service**.

Startup

Specifies whether the server attempts to start the specified service. Some services are always enabled and disregard this property if set. This setting is enabled by default.

Data type	Boolean
Default	true

Retries count

Specifies the number of times you want the file transfer service to retry sending or receiving a file after a communication failure occurs.

Data type	Integer
Default	3

If the retries count setting is blank, the file transfer service sets the default to 3. If the retries count setting is 0, the file transfer service does not retry. The default is the recommended value.

Retry wait time

Specifies the number of seconds that the file transfer service waits before it retries a failed file transfer.

Data type	Integer
Default	10

If the retry wait time setting is blank, the code sets the default to 10. If the retry wait time setting is 0, the file transfer service does not wait between retries. The default is the recommended value.

File synchronization service settings

Use this page to specify that a file set on one node matches that on the central deployment manager node and to ensure consistent configuration data across a cell.

You can synchronize files on individual nodes or throughout your system.

To view this administrative console page, click **System Administration > Node Agents > *node_agent_name* > File Synchronization Service**.

Startup

Specifies whether the server attempts to start the specified service. Some services are always enabled and disregard this property if set. This setting is enabled by default.

Data type	Boolean
Default	true

Synchronization Interval

Specifies the number of minutes that elapse between synchronizations. The default is 1 minute. Increase the time interval to synchronize files less often.

Data type	Integer
Units	Minutes
Default	1

The minimum value that the application server uses is 1. If you specify a value of 0, the application server ignores the value and uses the default of 1.

Automatic Synchronization

Specifies whether to synchronize files automatically after a designated interval. When this setting is enabled, the node agent automatically contacts the deployment manager every synchronization interval to attempt to synchronize the node's configuration repository with the master repository owned by the deployment manager.

Remove the checkmark from the check box if you want to control when files are sent to the node.

Data type	Boolean
------------------	---------

Default true

Startup Synchronization

Specifies whether the node agent attempts to synchronize the node configuration with the latest configurations in the master repository prior to starting an application server.

The default is to not synchronize files prior to starting an application server. Enabling the setting ensures that the node agent has the latest configuration but increases the amount of time it takes to start the application server.

Note that this setting has no effect on the startServer command. The startServer command launches a server directly and does not use the node agent.

Data type Boolean
Default false

Exclusions

Specifies files or patterns that should not be part of the synchronization of configuration data. Files in this list are not copied from the master configuration repository to the node, and are not deleted from the repository at the node.

The default is to have no files specified.

iSeries users: The default is */plugin-cfg.xml, which excludes the Web Server plug-in configuration file from synchronization.

You only need to specify files to exclude from synchronization when the synchronization is enabled. To specify a file, type in a file name and then press **Enter**. Each file name appears on a separate line.

Data type String
Units File names or patterns

Administrative agents: Resources for learning

Use the following links to find relevant supplemental information about WebSphere Application Server administrative agents and distributed administration. 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

- Listing of all IBM WebSphere Application Server Redbooks
- System Administration for WebSphere Application Server V5 -- Part 1: Overview of V5 Administration

Location service daemons

Location service daemons provide the CORBA location service in support of Remote Method Invocation and Internet Inter-ORB Protocol (RMI/IIOP).

When a client makes a remote call to an enterprise bean, a location service daemon determines which server or servers are eligible to process the request. The location service daemon makes the decision with the z/OS workload management function (WLM). The daemon then routes the request to the selected server, which establishes a CORBA session with the client. Subsequent calls to the same enterprise bean flow directly over the established session.

One location service daemon is available for each system in a cell. An example of a system is the z/OS operating system on a logical partition (LPAR). There are two options for daemon existence - a base application server environment and a network deployment manager environment. It is important to understand that in the creation of a Network Deployment configuration, a node goes through two life cycles: first, as a Base Application Server node; and then, when federated, a Network Deployment configuration node. In both phases of its life, the node relies upon the services of a Daemon, but the Daemon it relies upon is different before federation as compared to after.

All nodes must have access to the services of a Daemon. A Base Application Server node has its own Daemon. After federation the newly federated node will rely upon a different Daemon. What Daemon a federated node will use after federation is dependent on whether that node is on the same MVS image as the Deployment Manager or not.

Base Application Server node federated into Deployment Manager cell on same MVS image

When a Base Application Server node is federated into a Deployment Manager cell on the same MVS image, the Base Application Server's Daemon is set aside. The federated node starts using the Deployment Manager's Daemon. For example, when a Base Application Server node is federated into a Deployment Manager cell on the same MVS image, the already existing Deployment Manager Daemon simply picks up more work, while the original Base Application Server node's Daemon is no longer used.

A Base Application Server node's Daemon should be considered temporary. It is only used until the node is federated. Thereafter, the Base Application Server's Daemon is no longer used. That means the following things are temporary when a Base Application Server is intended for eventual federation into a Deployment Manager cell:

- Original Base Application Server Daemon's short name
- JOBNAME value Original Base Application Server Daemon's TCP ports

Base Application Server node federated into Deployment Manager cell from different MVS image

Things are slightly different when the Base Application Server node being federated is on a different MVS image from the Deployment Manager. The original Base Application Server Daemon is still set aside, but a new Daemon is created.

There are two points to be made here:

- Base Application Server node Daemons are temporary if that Base Application Server is federated. It doesn't matter if the Base Application Server is on the same MVS image or different from the Deployment Manager. The Daemon of any Base Application Server node intended for federation should be considered temporary.
- When a Base Application Server node on a different MVS image from the Deployment Manager is federated, the new Daemon that is created is identical to the Deployment Manager's Daemon: same TCP port values, same short name value, same JOBNAME value.

The Deployment Manager's Daemon should be considered a cell-wide resource. The short name and JOBNAME values should not contain a system-specific identifier since the same values may be used on multiple MVS images. And the TCP ports for all Daemons in a Network Deployment configuration will be the same.

Determining if the location service daemon is running.

Before starting either clusters or servers, you need to know if the location service daemon is running.

To determine if the location service daemon is running, issue one of the display commands from the MVS console.

You know the system is running if you see the address space BBODMN x running, where x is a letter (A,B,C, and so on).

Steps for stopping or canceling the location service daemon from the MVS console

If you cancel the location service daemon, it cancels all other application servers on the system.

Issue one of the following commands to stop the location service daemon:

```
STOP DAEMON01
CANCEL DAEMON01
CANCEL DAEMON01,ARMRESTART
```

Use the **cancel** command that has the ARMRESTART option if automatic restart management (ARM) is active and you want the ARM to restart the location service daemon.

Modifying location service daemon settings

The location service daemon is an integral component of the Remote Method Invocation and Internet Inter-ORB Protocol (RMI/IIOP) communication function for the WebSphere Application Server for z/OS. The daemon works with z/OS workload management to distribute RMI requests (for example enterprise bean requests) among application servers in a cell.

Before you begin

Complete the following items:

- Configure the location service daemon

You must first configure the location service daemon in the WebSphere customization dialogs. After you configure the daemon, you can modify or view the location service daemon settings from the administrative console.

- Optionally enable security

There is no specific setting to enable SSL for the location service daemon. If you want to use the Secure Sockets Layer (SSL) protocol to encrypt communications to the location service daemon, ensure that you complete the following items:

- Enable global security for the cell.
- Define a valid system SSL repertoire for the location service daemon.
- Set the z/OS user ID assigned to the location service daemon started task to the same z/OS user ID that was employed to create the key ring.

You can now modify the location service daemon settings in the administrative console.

1. Go to the Settings page for the location service daemon to view the help page.
2. Click **System Administration > z/OS Location Service Daemon** in the console navigation tree.
3. Specify the following values for the location service daemon:
 - The job name
 - The ports on which it listens
 - The host name or IP address
 - The SSL repertoire that it uses

The location service daemon settings have been modified.

z/OS location service daemon settings

Use this page to set the location service daemon for this cell.

Changes made to these settings apply to each location service daemon in a cell. One location service daemon is available on each system in a cell. An example of a system is the z/OS operating system on a logical partition (LPAR).

To view this administrative console page, click **System Administration > z/OS Location Service Daemon**.

Job name

Specifies the job name of the location service daemon. This name can be from one to eight characters. You can use alphanumeric or national language characters, (@#%).

Data type	String
Default	None

Host name

Specifies the host name or IP address to use when contacting the location service daemon.

Data type	String
Default	None

Port

Specifies the port on which the location service daemon listens for Remote Method Invocation and Internet Inter-ORB (RMI/IIOP) requests.

Because the system reserves port numbers less than 1024 for Transmission Control Protocol/Internet Protocol (TCP/IP) applications, the recommendation is to use ports numbers greater than 1023. However, the port range starts at 1.

Data type	Integer
Default	None
Range	1 to 65535

SSL port

Specifies the port on which the location service daemon listens for encrypted Remote Method Invocation and Internet Inter-ORB (RMI/IIOP) requests.

Because the system reserves port numbers less than 1024 for Transmission Control Protocol/Internet Protocol (TCP/IP) applications, the recommendation is to use ports numbers greater than 1023. However, the port range starts at 1.

Data type	Integer
Default	None
Range	1 to 65535

SSL Settings

Specifies a predefined list of Secure Sockets Layer settings for connections.

These settings are System SSL repertoires that you configured on the SSL repertoire panel. Select one repertoire from the list.

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

Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX
- AS/400
- CICS
- Cloudscape
- DB2
- DFSMS
- Domino
- Everyplace
- iSeries
- IBM
- IMS
- Informix
- iSeries
- Language Environment
- Lotus
- MQSeries
- MVS
- OS/390
- RACF
- Redbooks
- RMF
- SecureWay
- SupportPac
- Tivoli
- ViaVoice
- VisualAge
- VTAM
- WebSphere
- z/OS
- zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.