BEA WebLogic Server®

Developing Manageable Applications with JMX

BETA

Version 9.0 BETA
Revised: December 15, 2004

BETA

Copyright

Copyright © 2004 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

BEA

BETA

Contents

About This Document

1. Introduction and Roadmap

BETA

2. Understanding WebLogic Server MBeans

3. Accessing WebLogic Server MBeans with JMX

4. Managing a Domain's Configuration with JMX

BETA

BETA

BETA

About This Document

This document describes how to use the Sun Microsystems, Inc. Java Management Extensions (JMX) APIs to design and develop manageable applications.

Audience

This document is written mainly for J2EE application developers who are interested in using JMX to provide manage facilities for their applications beyond simple message logging. The document also provides information that is useful to software vendors who develop JMX-compatible management systems.

It is assumed that the reader is familiar with J2EE and general application management concepts.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

BETA

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that the user is told to enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Placeholders. *Example*: `String CustomerName;`

`UPPERCASE MONOSPACE TEXT`	Device names, environment variables, and logical operators.
	Examples:
	`LPT1`
	`BEA_HOME`
	`OR`

`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`

BETA

Convention	Usage
. . .	Indicates one of the following in a command line: • An argument can be repeated several times in the command line. • The statement omits additional optional arguments. • You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

BETA

CHAPTER 1

Introduction and Roadmap

BETA

As an application developer, you can greatly reduce the cost of operating and maintaining your applications by building in management facilities. The simplest facility is message logging, which reports events within your applications as they occur and writes messages to a file or other repository. Depending on the criticality of your application, the complexity of the production environment, and the types of monitoring systems your organization uses in its operations center, your needs might be better served by building richer management facilities based on Java Management Extensions (JMX). JMX enables a generic management system to monitor your application, raise notifications when the application needs attention, and change the configuration or runtime state of your application to remedy problems.

This document describes using JMX to make your applications manageable, and it describes creating management-aware applications that can configure themselves dynamically based on conditions within their operating environment.

The following sections describe the contents and organization of this guide—*Developing Manageable Applications with JMX*.

Document Scope and Audience

This document is a resource for software developers who develop management services for J2EE applications and for software vendors who develop JMX-compatible management systems. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server® or considering the use of JMX for a particular application.

The topics in this document are relevant during the design and development phases of a software project. This document does not address production phase administration, monitoring, or performance tuning topics. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-2.

It is assumed that the reader is familiar with J2EE and general application management concepts. This document emphasizes a hands-on approach to developing a limited but useful set of JMX management services. For information on applying JMX to a broader set of management problems, refer to the JMX specification or other documents listed in "Related Documentation" on page 1-2.

Guide to this Document

- This chapter, Introduction and Roadmap, introduces the organization of this guide.
- Chapter 2, "Understanding WebLogic Server MBeans," describes using JMX to manage WebLogic Server resources.
- Chapter 3, "Accessing WebLogic Server MBeans with JMX," describes how to access WebLogic Server MBeans from a JMX client.
- Chapter 4, "Managing a Domain's Configuration with JMX," describes how to manage a WebLogic Server domain's configuration through JMX.

Related Documentation

The Sun Developer Network includes a Web site that provides links to books, white papers, and additional information on JMX: http://java.sun.com/products/JavaManagement/.

To view the JMX 1.2 specification and API documentation, download it from http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html.

To view the JMX Remote API 1.0 specification and API documentation, download it from http://jcp.org/aboutJava/communityprocess/final/jsr160/index.html.

BETA

For guidelines on developing other types of management services for WebLogic Server applications, see the following documents:

- Using WebLogic Logging Services for Application Logging describes WebLogic support for internationalization and localization of log messages, and shows you how to use the templates and tools provided with WebLogic Server to create or edit message catalogs that are locale-specific.
- Understanding the WebLogic Diagnostic Service describes how system administrators can collect application monitoring data that has not been exposed through JMX, logging, or other management facilities.

For guidelines on developing and tuning WebLogic Server applications, see the following documents:

- Developing WebLogic Server Applications is a guide to developing WebLogic Server applications.
- WebLogic Server Performance and Tuning (not available for BETA) contains information on improving the performance of WebLogic Server applications.

Samples and Tutorials for the JMX Developer

In addition to this document, BEA Systems provides code samples and tutorials for JMX developers. The examples and tutorials illustrate management applications in action, and provide practical instructions on how to perform key JMX development tasks.

BEA recommends that you run some or all of the JMX examples before developing your own management applications.

Samples and tutorials for JMX are not available for BETA.

New and Changed JMX Features in This Release

Release 9.0 introduces several important changes to the WebLogic Server JMX implementation:

- "JMX 1.2 and JMX Remote API (JSR-160)" on page 1-4
- "Deprecated MBeanHome and Type-Safe Interfaces" on page 1-4
- "New Services for Accessing WebLogic Server MBeans" on page 1-7
- "Changes to the Model for Distributing Configuration Data in a Domain" on page 1-5
- "Changes to the MBean Hierarchy" on page 1-5

BETA

JMX 1.2 and JMX Remote API (JSR-160)

In 9.0, WebLogic Server upgrades its implementation of Java Management Extensions (JMX) from 1.0 to 1.2.

JMX 1.2 includes a new group of APIs that enable JMX components to communicate across JVMs (JSR-160). (See http://jcp.org/en/jsr/detail?id=160.) The JMX remote APIs are in the `javax.management.remote` package. The introduction of these APIs enables WebLogic Server to deprecate its proprietary `weblogic.management.RemoteMBeanServer` interface.

Note: As of this beta release of WebLogic Server 9.0, Sun's J2EE 1.5 documentation is not available to the public. However, you can view the JMX 1.2 API documentation from the J2SE 5.0 API Reference (in the `javax.management.*` packages). You can view the JSR-77 API documentation in the J2EE 1.4 API Reference (in the `javax.management.j2ee.*` packages).

Deprecated MBeanHome and Type-Safe Interfaces

Prior to 9.0, WebLogic Server supported a typed API layer over its JMX layer. Your JMX application classes could import type-safe interfaces for WebLogic Server MBeans, retrieve a reference to the MBeans through the `weblogic.management.MBeanHome` interface, and invoke the MBean methods directly.

As of 9.0, the `MBeanHome` interface is deprecated. Instead of using this API-like programming model, all JMX applications should use the standard JMX programming model, in which clients use the `javax.management.MBeanServerConnection` interface to discover MBeans, attributes, and attribute types at runtime. In this JMX model, clients interact indirectly with MBeans through the `MBeanServerConnection` interface.

If any of your classes import the type-safe interfaces (which are under `weblogic.management`), BEA recommends that you update to using the standard JMX programming model. See "Accessing WebLogic Server MBeans with JMX" on page 3-1.

BETA

Changes to the Model for Distributing Configuration Data in a Domain

In a WebLogic Server domain, the Administration Server is the central administration point for all other server instances (called Managed Servers).

Prior to 9.0, the Administration Server hosted a set of Administration MBeans which represented the persisted configuration for all servers and server resources in a domain. To enhance performance, each server instance replicated these MBeans locally and used the replicas, called Local Configuration MBeans. When a JMX client changed an Administration MBean, the Administration Server attempted to immediately update the Local Configuration MBeans on all server instances in the domain. In some cases, a Local Configuration MBean could not be updated without restarting a server instance and the replica and its master Administration MBean would contain different values. In addition, JMX clients could directly access Local Configuration MBeans and change their values, which also resulted in an inconsistent state between replica and master MBean.

In 9.0:

- Each server instance maintains a replica of the domain's `config.xml` file and uses the data in this file to instantiate configuration MBeans for its local resources. The local `config.xml` file and its MBean representation is read only and can be updated only through the distribution process described in the next paragraph.
- The Administration Server hosts a set of Edit MBeans which are the in-memory representation of all pending changes to a domain's configuration (Edit MBean data is backed up in a file called `pending/config.xml`). Changes in Edit MBeans do not take effect immediately. You must explicitly distribute them in a process that resembles a transaction. If any Managed Server is unable to consume a change, the entire set of changes in a distribution process is rolled back.

See "Managing a Domain's Configuration with JMX" on page 4-1.

Changes to the MBean Hierarchy

WebLogic Server organizes its MBeans in a hierarchical data model. For example, a WebLogic Server domain exposes its configuration data in a single MBean of type `DomainMBean`. The `DomainMBean.Servers` attribute contains the JMX object name for each instance of `ServerMBean`, which exposes configuration data for a specific server instance. The JMX object name for each MBean instance reflects the location of the MBean in the hierarchy: a child MBean's object name contains name/value pairs from the parent MBean's object name.

BETA

Prior to 9.0, JMX clients could create and access WebLogic Server MBeans by invoking `MBeanServer.createMBean` and passing a correctly constructed, hierarchical object name. However, if a JMX client incorrectly constructed the object name, the MBean would be created and registered but not recognized within the WebLogic Server data model.

As of 9.0, JMX clients must walk the MBean hierarchy to create, access, or destroy instances of WebLogic Server MBeans. All parent MBeans contain methods for creating and accessing their children, and there is no other option for creating child MBeans. For example, to create a server instance, a JMX client must retreive an object name handle to `DomainMBean` and invoke the `DomainMBean.createServer` method.

In addition to the parent-child relationship, some MBeans contain attributes that simply refer to other, related MBeans. For example, `ClusterMBean.Servers` contains the object names of all server instance that belong to the cluster, but `DomainMBean` is the parent of all instances of `ServerMBean`.

In this new 9.0 model, JMX clients no longer need to construct JMX object names. Instead, walk the hierarchy by successively invoking code similar to the following:

```
ObjectName on =
javax.management.MBeanServerConnection.getAttribute
        (object-name, attribute);
```

where:

- *`object-name`* is the object name of the current node (MBean) in the MBean tree.
- *`attribute`* is the name of an attribute in the current MBean that refers to another MBean.

To determine an MBean's location in an MBean tree, refer to the MBean's description in WebLogic Server MBean Reference. For each MBean, the *WebLogic Server MBean Reference* lists the parent MBean that contains the current MBean's factory methods. For an MBean whose factory methods are not public, the *WebLogic Server MBean Reference* lists other MBeans from which you can access the current MBean.

MBean Trees

Instead of organizing all MBeans in a domain into a single, large hierarchy, WebLogic Server divides its MBeans into different MBean trees:

- On each server instance, the server's configuration MBeans are in a single tree whose root is `DomainMBean`.

BETA

- On each server instance, the server's runtime MBeans are in a single tree whose root is `ServerRuntimeMBean`.
- On the Administration Server, MBeans for domain-wide services such as application deployment, JMS servers, and JDBC connection pools are in a single tree whose root is `DomainRuntimeMBean`.
- On the Administration Server, the Edit MBean Tree represents all pending changes to a domain's configuration.

Changes in Subsystem MBeans

Many subsystems, such as logging, JMS, JDBC, and deployment, have deprecated all or part of their old JMX interfaces and replaced them with new or updated MBeans.

See WebLogic Server MBean Reference, which lists all deprecated and new MBeans for 9.0.

New Services for Accessing WebLogic Server MBeans

A WebLogic Server domain maintains three types of MBean servers, each of which provides access to different MBean hierarchies. The Edit MBean server provides access to the hierarchy of editable configuration MBeans; the Domain Runtime MBean server provides federated access to all runtime MBeans and read-only configuration MBeans in the domain; and the Runtime MBean server provides access only to the runtime and read-only configuration MBeans on a specific server instance.

JMX clients use the standard `javax.remote.access` (JSR-160) APIs to access and interact with MBeans registered in the MBean servers.

Within each MBean server, WebLogic Server registers a service MBean under a simple object name. The attributes and operations in this MBean serve as your entry point into the MBean hierarchies and enable JMX clients to navigate to all WebLogic Server MBeans in an MBean server after supplying only a single object name.

See "MBean Servers" on page 2-13.

No Support for Registering Custom MBeans

As of this 9.0 Beta release, there is no support for registering an MBean that you have created to manage your applications or resources.

BETA

WebLogic Server does not expose its `MBeanServer` interface through the JNDI tree; only the service interfaces are exposed, and these services only provide access to WebLogic Server MBeans.

BEA will enable you to register custom MBeans in a subsequent release during the Beta phase of 9.0.

New Reference Document for WebLogic Server MBeans

All public WebLogic Server MBeans are described in a new document, WebLogic Server MBean Reference. For each MBean, the document describes:

- The MBean's factory methods and other points of access within WebLogic Server MBean trees
- The data type, read-write privileges, and other information for each attribute
- The parameters, signature, and other information for each operation

BETA

CHAPTER 2

Understanding WebLogic Server MBeans

BETA

WebLogic Server® provides its own set of MBeans that you can use to configure, monitor, and manage WebLogic Server resources. The following sections describe how WebLogic Server distributes and maintains its MBeans:

"WebLogic Server MBean Reference" provides a detailed reference for all WebLogic Server MBeans.

Basic Organization of a WebLogic Server Domain

A WebLogic Server administration **domain** is a logically related group of WebLogic Server resources. Domains include a special WebLogic Server instance called the **Administration Server**, which is the central point from which you configure and manage all resources in the domain. Usually, you configure a domain to include additional WebLogic Server instances called **Managed Servers**. You deploy Web applications, EJBs, and other resources onto the Managed Servers and use the Administration Server for configuration and management purposes only.

Using multiple Managed Servers enables you to balance loads and provide failover protection for critical applications, while using single Administration Server simplifies the management of the

Managed Server instances. For more information about domains, refer to "Understanding WebLogic Server Domains" in *Configuring and Managing WebLogic Server*.

Separate MBean Types for Monitoring and Configuring

All WebLogic Server MBeans can be organized into one of the following general types based on whether the MBean monitors or configures servers and resources:

- **Runtime MBeans** contain information about the runtime state of servers and resources. Because Runtime MBeans contain only transient data, they do not persist their data. When you shut down a server instance, all runtime statistics and metrics from the Runtime MBeans are destroyed.
- **Configuration MBeans** contain information about the configuration of servers and resources. The data in these MBeans comes from the domain's XML configuration files and therefore is reconstituted after you shut down and restart a server.

  The Configuration MBeans and configuration files that represent the working (active) configuration of a domain are read only. To enable JMX clients to modify a domain's configuration, the Administration Server maintains a separate set of editable, pending Configuration MBeans and configuration files. As part of the WebLogic Server change management process, a JMX client modifies a pending Configuration MBean, saves the changes to the pending configuration files, and then distributes the pending configuration files to all servers in the domain. If all servers are able to consume the changes in the pending files, then the pending files become the active configuration files and the active Configuration MBeans are updated to reflect the new configuration values.

BETA

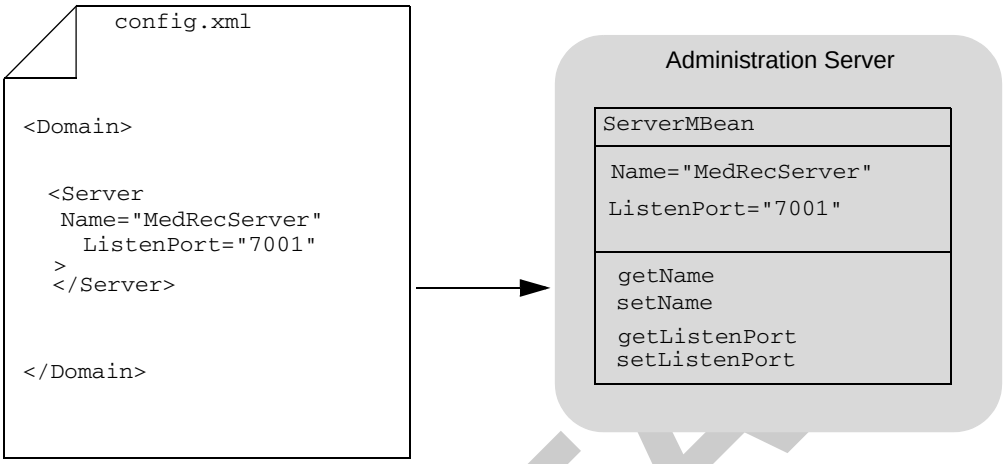
The Life Cycle of WebLogic Server MBeans

The life cycle of a Runtime MBean follows that of the resource for which it exposes runtime data. For example, when you start a server instance, the server instantiates a `ServerRuntimeMBean` and populates it with the current runtime data. Each resource updates the data in its Runtime MBean as its state changes. The resource destroys its Runtime MBeans when it is stopped.

The life cycle of a Configuration MBean is slightly more complicated:

1. When you start the Administration Server, the server initializes the Configuration MBeans for itself and its resources with data from the domain's `config.xml` file. (See Figure 2-1.)

Figure 2-1 Initializing Configuration MBeans on Administration Server

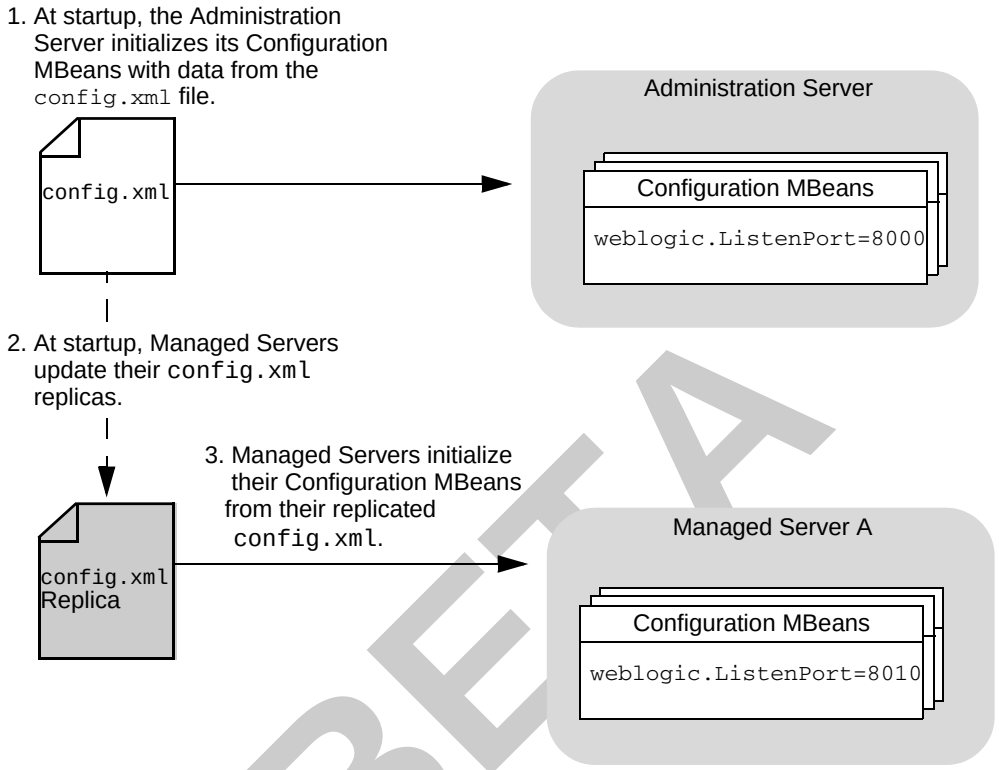


2. The Administration Server initializes the pending Configuration MBeans for all servers and resources in the domain. The data for these MBeans comes from one of the following sets of files:
   - If all changes in the pending configuration files were successfully activated before the Administration Server was shut down, pending Configuration MBeans are initialized with data from the active configuration files.
   - If a user changed the value of a pending Configuration MBean and saved the changes but the Administration Server was shut down before the pending changes were activated, pending Configuration MBeans are initialized with data from the pending configuration files.

   Unlike the life cycle of active Configuration MBeans, which follows the life cycle of their host server instance, the life cycle of pending Configuration MBeans is tied to the Administration Server. For example, when you shut down Managed Server A, the Configuration MBeans that it hosts are destroyed, but its pending Configuration MBeans on the Administration Server are not destroyed until the Administration Server shuts down.
3. When you start a Managed Server, the server synchronizes its active configuration files with the Administration Server's files and then creates the Configuration MBeans for itself and its resources using the configuration from its active `config.xml` file. (See Figure 2-3.)

BETA

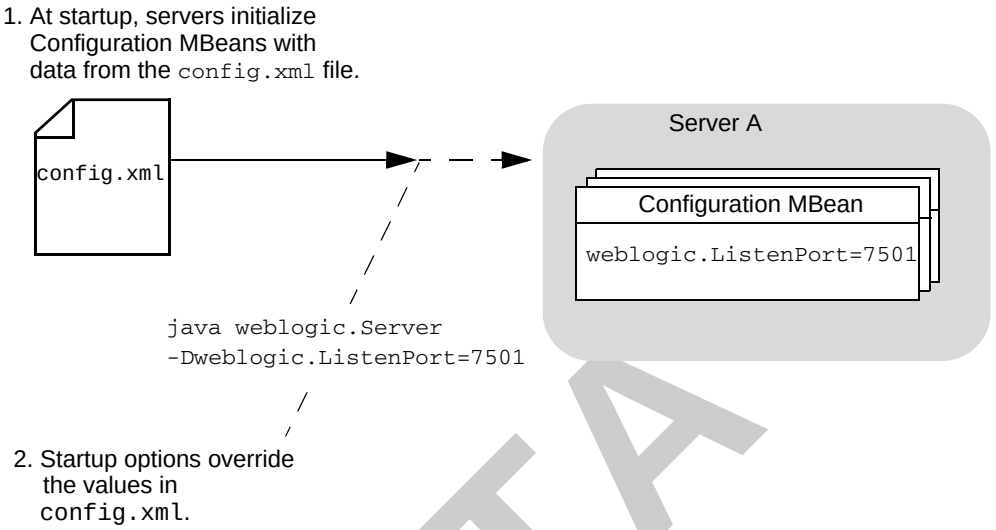
Figure 2-2 Initializing Configuration MBeans on Managed Servers



BETA

Arguments in a server's startup command override values from the `config.xml` file. For example, for Server A, the `config.xml` file states that its listen port is 8010. When you use the `weblogic.Server` command to start Server A, you include the `-Dweblogic.ListenPort=7501` startup option to change the listen port for the current server session. The server instance initializes its Configuration MBeans from the `config.xml` file but substitutes `7501` as the value of its listen port. When you restart Server A, it will revert to using the value from the `config.xml` file, `8010`. (See Figure 2-3.)

Figure 2-3 Overriding config.xml Values



4. For information on how configuration changes are activated in a domain, see "Managing a Domain's Configuration with JMX" on page 4-1.
5. Configuration MBeans are destroyed when you shut down the server instance that hosts them.

Configuration MBeans and Managed Server Independence

Managed Server Independence (MSI) is a feature that enables a Managed Server to start if the Administration Server is unavailable. If a Managed Server is configured for MSI, in addition to instantiating its own Configuration MBeans, it instantiates a copy of all Configuration MBeans for all servers in the domain. This replication of MBeans enable WebLogic Server resources to communicate with one another across server instances.

The Configuration MBeans on each Managed Server are exactly the same because the `config.xml` file that each Managed Server uses is an exact replica of the other. Because you cannot modify the configuration of a domain if the Administration Server is unavailable, the Configuration MBeans across the domain remain unchanged.

For more information on MSI, refer to "Starting a Managed Server When the Administration Server Is Not Accessible" in *Configuring and Managing WebLogic Server*.

BETA

WebLogic Server MBean Data Model

WebLogic Server organizes its Runtime and Configuration MBeans in a hierarchical data model. For example, a WebLogic Server domain exposes its configuration data in a single MBean of type `DomainMBean`. Each server in the domain exposes its configuration data in an MBean of type `ServerMBean`. In the WebLogic Server data model, each `ServerMBean` is a child of the `DomainMBean`.

The data model is expressed through the following features:

- "Factory Methods" on page 2-6
- "Containment Attributes and Lookup Operations" on page 2-7
- "WebLogic Server MBean Object Names" on page 2-7
- "Configuration, Runtime, and Domain-Runtime Hierarchies" on page 2-11

Factory Methods

Parent MBeans contain factory methods for child MBeans and JMX clients must invoke these factory methods to create or destroy MBeans. Clients cannot use `javax.management.MBeanServer.create()` or `register()` to create instances of WebLogic Server MBeans. WebLogic Server imposes this restriction on the JMX specification to maintain the integrity of its data model. Without this restriction, a JMX client could register a WebLogic Server MBean under a name that the MBean's parent would not recognize and would not consider as part of the WebLogic Server MBean hierarchy.

Note: To create or destroy custom MBeans (MBeans you have created to manage your applications), use the standard `MBeanServer.create()` or `register()` methods. Custom MBeans are not part of the WebLogic Server data model and are not subject to its factory method restrictions.

In some cases, an MBean's factory methods are not public because of dependencies within a server instance. In these cases the parent manages the life cycle of its children. For example, each `ServerMBean` must have one and only one child `LogMBean` to configure the server's local log file. The factory methods for `LogMBean` are not public, and `ServerMBean` maintains the life cycle of its `LogMBean`.

BETA

Containment Attributes and Lookup Operations

All parent MBeans include attributes that contain their children. You can use these child containment attributes to get object names for child MBeans. Then you use the child's object name in standard JMX APIs to get or set values of the child MBean's attributes or invoke its methods. For example, `DomainMBean` includes a `Servers` attribute which contains an array of `ServerMBean` objects. To get object names for all instances of `ServerMBean` in the domain, you get the value of the `Servers` attribute and cast the array as `javax.management.ObjectName[]`. For each object name in the array, you can use `javax.management.MBeanServerConnection.getAttribute(` `ObjectName` *`ServerMBean-object-name`* `String` *`ServerMBean-attribute-name`*`)` to get the value of *`Server-MBean-attribute-name`*.

Some WebLogic Server MBeans include containment attributes for MBeans that are not children, but that are otherwise logically related. For example, `ClusterMBean` is not the parent of `ServerMBean`, it includes a `Servers` attribute that contains a reference to all `ServerMBean` instances that are part of the cluster.

If you know the name that was used to create a specific server or resource, you can use a lookup operation in the parent MBean to get the object name. For example, `DomainMBean` includes an operation named `lookupServers(String` *`name`*`)`, which takes as a parameter the name that was used to create a server instance. If you named a server `MS1`, you could pass a `String` object that contains `MS1` to the `lookupServers` method and the method would return the object name for `MS1`.

BETA

WebLogic Server MBean Object Names

All MBeans must be registered in an MBean server under an object name of type `javax.management.ObjectName`. WebLogic Server follows a convention in which object names for child MBeans contain part of its parent MBean object name.

Note: If you learn the WebLogic Server naming conventions, you can understand where an MBean instance resides in the data hierarchy by observing its object name. However, if you use containment attributes or lookup operations to get object names for WebLogic Server MBeans, your JMX applications do not need to construct or parse object names.

WebLogic Sever naming conventions encode its MBean object names as follows:

`domain``:Name=`*`name`*`,Type=`*`type`*`[,`*`TypeOfParentMBean`*`=`*`NameOfParentMBean`*`]`
`[,`*`TypeOfParentMBean1`*`=`*`NameOfParentMBean1`*`]...`

where:

- *domain*: is a case-sensitive string that defines a top level within the JMX namespace (the JMX domain name).

  For WebLogic Server MBeans, the JMX domain name is the name of the WebLogic Server domain in which the MBean resides.

  For example, in a WebLogic Server domain named mydomain, all WebLogic Server MBean names start with the `mydomain:` string and therefore are in the `mydomain` JMX domain. If you create custom MBeans for your applications, you can add them to the `mydomain:` JMX domain or create your own JMX domain.

- `Name=`*name*`,Type=`*type*`[,`*TypeOfParentMBean*`=`*NameOfParentMBean*`][,`*TypeOfParentMBean1*`=`*NameOfParentMBean1*`]...` is a set of JMX key properties.

The order of the key properties is not significant, but the name must begin with *domain*:.

BETA

Table 2-1 describes the key properties that WebLogic Server encodes in its MBean object names.

Table 2-1 WebLogic Server MBean Object Name Key Properties

This Key Property	Specifies
`Name=`*`name`*	The string that you provided when you created the resource that the MBean represents. For example, when you create a server, you must provide a name for the server, such as MS1. The `ServerMBean` that represents MS1 uses `Name=MS1` in its JMX object name. If you create an MBean, you must specify a value for this `Name` component that is unique amongst all other MBeans in a domain.
`Type=`*`type`*	Refers to the type of MBean. To determine the value that you provide for the `Type` component, find the MBean's type name and remove the `MBean` suffix from the class name. For example, for an MBean that is an instance of the `ServerRuntimeMBean`, use `ServerRuntime`.

BETA

Table 2-1 WebLogic Server MBean Object Name Key Properties

This Key Property	Specifies

TypeOfParentMBean=*NameOfParentMBean*	To create a hierarchical namespace, WebLogic Server MBeans use one or more instances of this attribute in their object names. The levels of the hierarchy are used to indicate scope. For example, a `LogMBean` at the domain level of the hierarchy manages the domain-wide message log, while a `LogMBean` at a server level manages a server-specific message log.
	
	WebLogic Server child MBeans with implicit creator methods use the same value for the `Name` component as the parent MBean. For example, the `LogMBean` that is a child of the `MedRecServer Server` MBean uses `Name=MedRecServer` in its object name:
	
	`medrec:Name=MedRecServer,Type=Log,Server=MedRecServer`
	
	WebLogic Server cannot follow this convention when a parent MBean has multiple children of the same type.
	
	Some MBeans use multiple instances of this component to provide unique identification. For example, the following is the object name for an `EJBComponentRuntime` MBean for in the MedRec sample application:
	`medrec:ApplicationRuntime=MedRecServer_MedRecEAR,`
	`Name=MedRecServer_MedRecEAR_Session`
	`EJB,ServerRuntime=MedRecServer,Type=EJBComponentRuntime`
	
	The `ApplicationRuntime=MedRecServer_MedRecEAR` key property indicates that the EJB instance is a module within the MedRec enterprise application and a child of the `MedRecServer_MedRecEAR` ApplicationRuntimeMBean. The `ServerRuntime=MedRecServer` key property indicates that the EJB instance is currently deployed on a server named MedRecServer and a child of the `MedRecServer` ServerRuntimeMBean.

`Location=`*servername*	When you access Runtime MBeans or Configuration MBeans through the Domain Runtime MBean server, the MBean object names include a `Location=`*servername* key property which specifies the name of the server instance on which that MBean is located. See "MBean Servers" on page 2-13. Singleton MBeans, such as `DomainRuntimeMBean` and `ServerLifeCycleRuntimeMBean` exist only on the Administration Server and do not need to include this key property.

BETA

Configuration, Runtime, and Domain-Runtime Hierarchies

Instead of organizing all MBeans in a domain into a single, large hierarchy, WebLogic Server divides its MBeans into the following hierarchies (see Figure 2-4):

- Active Configuration MBeans are in a single hierarchy whose root is `DomainMBean`. (See DomainMBean in *WebLogic Server MBean Reference*.)

  Below this root are MBeans such as:

  - `ClusterMBean`
  - `ServerMBean`
  - `ApplicationMBean`
  - `RealmMBean`

- Runtime MBeans are in a single hierarchy whose root is `ServerRuntimeMBean`. (See ServerRuntimeMBean in *WebLogic Server MBean Reference*.)

  Below this root are MBeans such as:

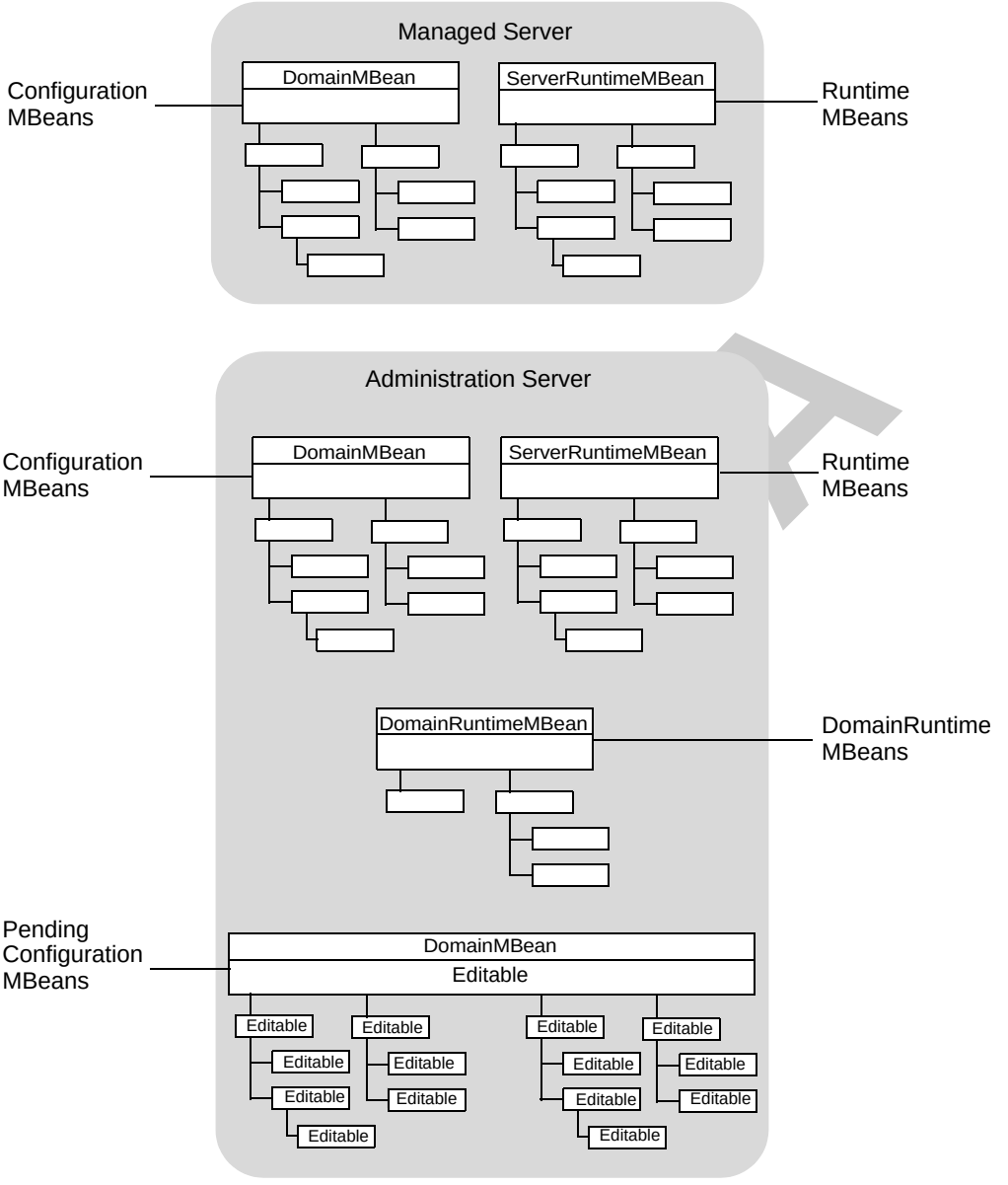
  - `ClusterRuntimeMBean`
  - `ApplicationRuntimeMBean`
  - `JDBCResourceRuntimeMBean`
  - `JMSRuntimeMBean`

- On the Administration Server, MBeans for domain-wide services such as application deployment, JMS servers, and JDBC connection pools are in a single hierarchy whose root is `DomainRuntimeMBean`. (See DomainRuntimeMBean in *WebLogic Server MBean Reference*.)

- On the Administration Server, the pending Configuration MBeans are in a single hierarchy whose root is `DomainMBean`. This hierarchy contains an editable copy of all Configuration MBeans in the domain and it is used only as part of the change management process. See "Managing a Domain's Configuration with JMX" on page 4-1.

BETA

Figure 2-4 WebLogic Server MBean Hierarchies



MBean Servers

At the core of any JMX agent is the MBean server, which creates, registers, and provides access to MBeans. A WebLogic Server domain maintains three types of MBean servers, each of which fulfills a specific function. Table 2-2 describes each type of MBean server.

Table 2-2 Types of MBean Servers in a WebLogic Server Domain

This MBean server type	Creates, registers, and provides access to...
Domain Runtime MBean server	• The Domain Runtime MBean hierarchy, which contains MBeans for domain-wide services such as application deployment, JMS servers, and JDBC connection pools. This MBean server also registers proxy object names for all Runtime MBeans and activated Configuration MBeans in the domain. The proxies enable JMX clients to access the following from the Domain Runtime MBean server: • A Runtime MBean hierarchy that contains all Runtime MBeans for all servers in the domain. • A Configuration MBean hierarchy that contains all activated Configuration MBeans for all servers in the domain. The process of registering proxies and forwarding requests is invisible to JMX clients that use containment attributes or lookup operations to navigate the hierarchies. If your JMX client accesses WebLogic Server MBeans by constructing object names, the client must add a `Location=`*`servername`* key property to the MBean object name. See "WebLogic Server MBean Object Names" on page 2-7. Only the Administration Server hosts an instance of this MBean server.
Runtime MBean server	• The hierarchy of the Runtime MBeans that are on a single server instance. • The hierarchy of the activated Configuration MBeans that are on a single server instance. Each server in the domain hosts an instance of this MBean server.
Edit MBean server	• The hierarchy of pending Configuration MBeans. Only the Administration Server hosts an instance of this MBean server.

WebLogic Server registers these MBean servers in the JNDI tree. (See Table 3-1, "JNDI Names for WebLogic MBean Servers," on page 3-2.)

BETA

The MBeanServerConnection Interface

To access the MBeans that are registered in a WebLogic MBean server, JMX clients must use the `javax.management.MBeanServerConnection` interface. This interface contains a standard set of methods for getting or setting MBean attributes and invoking MBean operations. See `MBeanServerConnection` in the J2SE 5.0 API Reference.

WebLogic MBean servers themselves do not provide public APIs; JMX clients must use the `MBeanServerConnection` interface.

Service MBeans

Within each MBean server, WebLogic Server registers a service MBean under a simple object name. The attributes and operations in this MBean serve as your entry point into the MBean hierarchies and enable JMX clients to navigate to all WebLogic Server MBeans in an MBean server after supplying only a single object name. (See "Accessing WebLogic Server MBeans with JMX" on page 3-1.)

JMX clients that do not use the entry point (service) MBean must correctly construct an MBean's object name to get and set the MBean's attributes or invoke its operations. Because the object names must be unique, they are usually long and difficult to construct from a client.

Table 2-3 describes each type of service MBean.

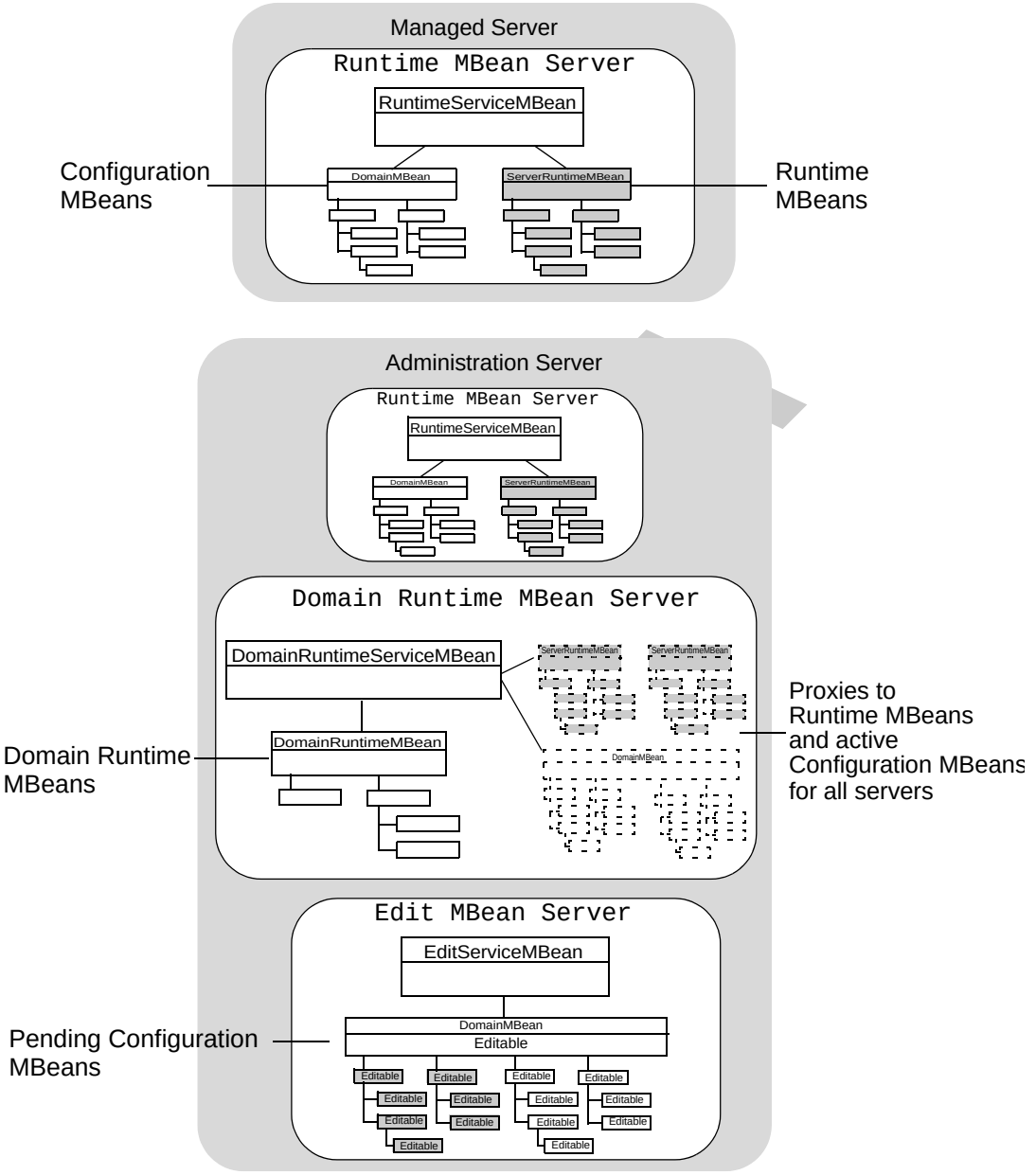
Table 2-3 Service MBeans

MBean	Registered in...
`DomainRuntimeServiceMBean`	The Domain Runtime MBean server See DomainRuntimeServiceMBean in *WebLogic Server MBean Reference*.
`RuntimeServiceMBean`	Runtime MBean servers See RuntimeServiceMBean in *WebLogic Server MBean Reference*.
`EditServiceMBean`	The Edit MBean server See EditServiceMBean in *WebLogic Server MBean Reference*.

Figure 2-5 illustrates how the MBean servers and service MBeans are distributed within a domain.

BETA

Figure 2-5 MBean Servers and Service MBeans



BETA

CHAPTER 3

Accessing WebLogic Server MBeans with JMX

BETA

The following sections describe how to access WebLogic Server MBeans from a JMX client:

Set Up the Classpath for Remote Clients

If your JMX client runs in its own JVM (that is, a JVM that is not a WebLogic Server instance), you must include the following JAR files in the client's classpath to access WebLogic Server MBeans:

- The `rt.jar` for a JRE that is based on JDK 1.5 or greater.

  For example, `C:\jdk1.5.0\jre\lib\rt.jar`

  Prior to JDK 1.5, JMX classes were not part of the runtime environment.

- `BEA_HOME\weblogic90\server\lib\wlclient.jar`

  The WebLogic Server client JAR contains classes needed to connect to the WebLogic Server service interfaces.

Connect to an MBean Server

Each WebLogic Server domain includes three types of MBean servers, each of which provides access to different MBean hierarchies. See "MBean Servers" on page 2-13.

To connect to a WebLogic MBean server:

1. Describe the address of the MBean server by constructing a `javax.management.remote.JMXServiceURL` object.

   Pass the following parameter values to the constructor (see the J2SE Javadoc for `JMXServiceURL`):

   – `"t3"` as the protocol for communicating with the MBean server
   – Listen address of the WebLogic Server instance that hosts the MBean server
   – Listen port of the WebLogic Server instance
   – Absolute JNDI name of the MBean server. The JNDI name must start with `/jndi/` and be followed by one of the JNDI names described in Table 3-1.

Table 3-1 JNDI Names for WebLogic MBean Servers

MBean Server	JNDI Name
Domain Runtime MBean server	`weblogic.management.mbeanservers.domainruntime`
Runtime MBean server	`weblogic.management.mbeanservers.runtime`
Edit MBean server	`weblogic.management.mbeanservers.edit`

2. Construct a `javax.management.remote.JMXConnector` object. This object contains methods that JMX clients use to connect to MBean servers.

   The constructor method for `JMXConnector` is:
   `javax.management.remote.JMXConnectorFactory.`
   `connector(JMXServiceURL` *`serviceURL`*`, Map<String,?>` *`environment`*`)`

   Pass the following parameter values to the constructor (see the J2SE Javadoc for `JMXConnectorFactory`):

   – The `JMXServiceURL` object you created in the previous step.
   – A hashmap that contains the following name-value pairs:

     `javax.naming.Context.SECURITY_PRINCIPAL,` *`admin-user-name`*

BETA

```
javax.naming.Context.SECURITY_CREDENTIALS, admin-user-password

javax.management.remote.JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAG
ES, "weblogic.management.remote"
```

The `weblogic.management.remote` package defines the protocols that can be used to connect to the WebLogic MBean servers. Remote JMX clients must include these WebLogic Server on their classpath. See "Set Up the Classpath for Remote Clients" on page 3-1.

3. Connect to the WebLogic MBean server by invoking the `JMXConnector.getMBeanServerConnection()` method.

   The method returns an object of type `javax.management.MBeanServerConnection`.

   The `MBeanServerConnection` object is your connection to the WebLogic MBean server. You can use it for local and remote connections. See the J2SE Javadoc for `MBeanServerConnection`.

4. BEA recommends that when your client finishes its work, close the connection to the MBean server by invoking the `JMXConnector.close()` method.

Example: Connecting to the Domain Runtime MBean Server

Note the following about the code in Listing 3-1:

- The class uses global variables, `connection` and `connector`, to represent the connection to the MBean server. The `initConnection()` method, which assigns the value to the `connection` and `connector` variables, should be called only once per class instance to establish a single connection that can be reused within the class.
- The `initConnection()` method takes the username and password (along with the server's listen address and listen port) as arguments that are passed when the class is instantiated. BEA recommends this approach because it prevents your code from containing unecrypted user credentials. The `String` objects that contain the arguments will be destroyed and removed from memory by the JVM's garbage collection routine.
- When the class finishes its work, it invokes `JMXConnector.close()` to close the connection to the MBean server. (See the J2SE Javadoc for `JMXConnector`.)

Listing 3-1 Connecting to the Domain Runtime MBean Server

```
public class MyConnection {
```

BETA

```
   private static MBeanServerConnection connection;
   private static JMXConnector connector;
   private static final ObjectName service;

   /*
   * Initialize connection to the Domain Runtime MBean server.
   */
   public static void initConnection(String hostname, String portString,
      String username, String password) throws IOException,
      MalformedURLException {

      String protocol = "t3";
      Integer portInteger = Integer.valueOf(portString);
      int port = portInteger.intValue();
      String jndiroot = "/jndi/";
      String mserver = "weblogic.management.mbeanservers.domainruntime";

      JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname, port,
      jndiroot + mserver);

      Hashtable h = new Hashtable();
      h.put(Context.SECURITY_PRINCIPAL, username);
      h.put(Context.SECURITY_CREDENTIALS, password);
      h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
         "weblogic.management.remote");
         connector = JMXConnectorFactory.connect(serviceURL, h);
         connection = connector.getMBeanServerConnection();
   }

   public static void main(String[] args) throws Exception {
      String hostname = args[0];
      String portString = args[1];
      String username = args[2];
      String password = args[3];

      MyConnection c= new MyConnection();
      initConnection(hostname, portString, username, password);
...
      connector.close();
   }
}
```

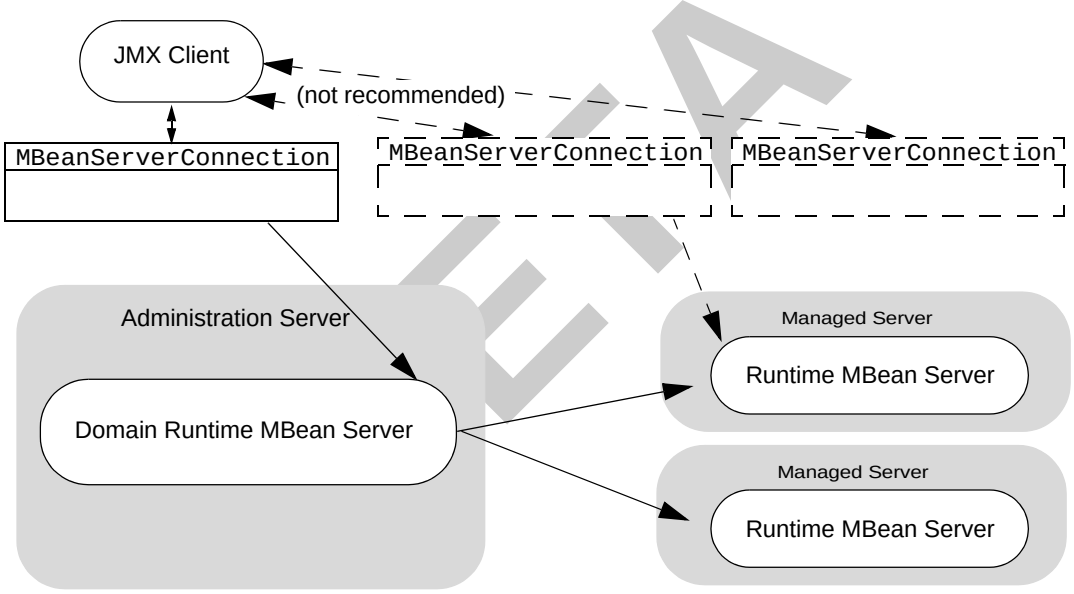
BETA

Best Practices: Domain Runtime MBean Server versus Runtime MBean Server

If your JMX client monitors WebLogic Server MBeans, then the client must use either the Domain Runtime MBean server or Runtime MBean server to access MBeans.

If your client reads MBean values on multiple server instances, or if your client runs in a separate JVM, BEA recommends that you connect to the Domain Runtime MBean server on the Administration Server instead of connecting separately to each Runtime MBean server on each server instance in the domain.

Figure 3-1 Domain Runtime MBean Server versus Runtime MBean Server



In general, code that uses the Domain Runtime MBean server is easier to maintain and is more secure for the following reasons:

- Your code only needs to construct a single URL for connecting to the Domain Runtime MBean server on the Administration Server. Thereafter, the code can look up values for all server instances and optionally filter the results.

If your code uses the Runtime MBean server to read MBean values on multiple server instances, it must constuct a URL for each server instance, each of which has a unique listen address/listen port combination.

- You can route all administrative traffic in a WebLogic Server domain through the Administration Server's secured administration port, and you can use a firewall to prevent connections to Managed Server administration ports from outside the firewall.

The tradeoff for directing all JMX requests through the Administration Server is a slight degradation in performance due to network latency. Connecting directly to each Managed Servers's Runtime MBean server to read MBean values eliminates the network hop that the Domain Runtime MBean server makes to retrieve a value from a Managed Server. However, for most network topologies and performance requirements, the simplified code maintenance and enhanced security that the Domain Runtime MBean server enables is preferable.

Navigate MBean Hierarchies

WebLogic Server organizes its MBeans in a hierarchical data model. (See "WebLogic Server MBean Data Model" on page 2-6.) In this model, all parent MBeans include attributes that contain their children. You can use these child containment attributes to get object names for child MBeans. Then you use the child's object name in standard JMX APIs to get or set values of the child MBean's attributes or invoke its methods.

To navigate the WebLogic Server MBean hierarchy:

1. Initiate a connection to an MBean server.

   See the previous section, "Connect to an MBean Server" on page 3-2.

   Initiating the connection returns an object of type `javax.management.MBeanServerConnection`.

2. Obtain the object name for an MBean at the root of an MBean hierarchy by invoking the `MBeanServerConnection.getAttribute(ObjectName` *`object-name`*`, String` *`attribute`*`)` method where:

   – *`object-name`* is the object name of the service MBean that is registered in the MBean server. (See "Service MBeans" on page 2-14.)

   Table 3-2 describes the type of service MBeans that are available in each type of MBean server.

BETA

– *attribute* is the name of a service MBean attribute that contains the root MBean.

Table 3-2 Service MBeans

MBean Server	Service MBean	JMX object name:
The Domain Runtime MBean server	`DomainRuntimeServiceMBean` For a list of root MBeans that this service MBeans can access, see DomainRuntimeServiceMBean in *WebLogic Server MBean Reference.*	`weblogic:Name=DomainRuntimeService`
Runtime MBean servers	`RuntimeServiceMBean` For a list of root MBeans that this service MBeans can access, see RuntimeServiceMBean in *WebLogic Server MBean Reference*.	`weblogic:Name=RuntimeService`
The Edit MBean server	`EditServiceMBean` For a list of root MBeans that this service MBeans can access, see EditServiceMBean in *WebLogic Server MBean Reference*.	`weblogic:Name=EditService`

3. Successively invoke code similar to the following:
   `ObjectName on = MBeanServerConnection.getAttribute(`*`object-name`*`, `*`attribute`*`)`
   where:
   – *object-name* is the object name of the current node (MBean) in the MBean hierarchy.
   – *attribute* is the name of an attribute in the current MBean that contains one or more instances of a child MBean. If the attribute contains multiple children, assign the output to an object name array, `ObjectName[]`.

To determine an MBean's location in an MBean tree, refer to the MBean's description in WebLogic Server MBean Reference. For each MBean, the *WebLogic Server MBean Reference* lists the parent MBean that contains the current MBean's factory methods. For an MBean whose factory methods are not public, the *WebLogic Server MBean Reference* lists other MBeans from which you can access the current MBean.

Example: Getting the Name and State of Servers

The code example in Listing 3-2 connects to the Domain Runtime MBean server and uses the `DomainRuntimeServiceMBean` to get the object name for each `ServerRuntimeMBean` in the

BETA

domain. Then it retrieves the value of each server's `ServerRuntimeMBean` `Name` and `State` attributes.

Note the following about the code in Listing 3-2:

- In addition to the `connection` and `connector` global variables, the class assigns the object name for the WebLogic Server service MBean to a global variable. Methods within the class will use this object name frequently, and once it is defined it does not need to change.
- The `getServerRuntimes()` method gets the value of the `DomainRuntimeServiceMBean` `ServerRuntimes` attribute, which contains an array of all `ServerRuntimeMBean` instances in the domain. (See DomainRuntimeServiceMBean in *WebLogic Server MBean Reference*.)

  Because `ServerRuntimeMBean` is the root of a server's runtime MBean tree, any JMX client that retrieves values from this tree will need to use a method similar to `getServerRuntimes()`. (See ServerRuntimeMBean in *WebLogic Server MBean Reference*.)

Listing 3-2 Example: Get the Name and State of Servers

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Hashtable;

import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

public class GetServerState {

   private static MBeanServerConnection connection;
   private static JMXConnector connector;
   private static final ObjectName service;

   // Initializing the object name for DomainRuntimeServiceMBean
   // so it can be used throughout the class.
   static {
```

BETA

```
   try {
      service = new ObjectName("weblogic:Name=DomainRuntimeService");
   }catch (MalformedObjectNameException e) {
      throw new AssertionError(e.getMessage());
   }
}

/*
* Initialize connection to the Domain Runtime MBean server
*/
public static void initConnection(String hostname, String portString,
   String username, String password) throws IOException,
   MalformedURLException {
   String protocol = "t3";
   Integer portInteger = Integer.valueOf(portString);
   int port = portInteger.intValue();
   String jndiroot = "/jndi/";
   String mserver = "weblogic.management.mbeanservers.domainruntime";
   JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname,
      port, jndiroot + mserver);
   Hashtable h = new Hashtable();
   h.put(Context.SECURITY_PRINCIPAL, username);
   h.put(Context.SECURITY_CREDENTIALS, password);
   h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
      "weblogic.management.remote");
   connector = JMXConnectorFactory.connect(serviceURL, h);
   connection = connector.getMBeanServerConnection();
}

/*
* Get an array of ServerRuntimeMBeans.
* This MBean is the root of the Runtime MBean hierarchy, and
* each server in the domain hosts its own instance.
*/
public static ObjectName[] getServerRuntimes() throws Exception {
   return (ObjectName[]) connection.getAttribute(service,
      "ServerRuntimes");
}
```

BETA

```
    /*
    * Iterate through ServerRuntimeMBeans and get the name and state
    */
    public void getNameAndState() throws Exception {
       ObjectName[] serverRT = getServerRuntimes();
       System.out.println("got server runtimes");
       int length = (int) serverRT.length;
       for (int i = 0; i < length; i++) {
          String name = (String) connection.getAttribute(serverRT[i],
             "Name");
          String state = (String) connection.getAttribute(serverRT[i],
             "State");
          System.out.println("Server name: " + name + ".   Server state: "
             + state);
       }
    }

    public static void main(String[] args) throws Exception {
       String hostname = args[0];
       String portString = args[1];
       String username = args[2];
       String password = args[3];

       GetServerState s = new GetServerState();
       initConnection(hostname, portString, username, password);
       s.getNameAndState();
       connector.close();
    }
}
```

BETA

Example: Monitoring Servlets

Each servlet in a Web application provides instance of `ServletRuntimeMBean` which contains information about the servlet's runtime state. (See ServletRuntimeMBean in *WebLogic Server MBean Reference*.)

In the WebLogic Server data model, the path to a `ServletRuntimeMBean` is as follows:

1. The Domain Runtime MBean server (for all servlets on all servers in the domain), or the Runtime MBean server on a specific server instance.
2. `DomainRuntimeServiceMBean` or `RuntimeServiceMBean`, `ServerRuntimes` attribute.
3. `ServerRuntimeMBean`, `ApplicationRuntimes` attribute.
4. `ApplicationRuntimeMBean`, `ComponentRuntimes` attribute.

   The `ComponentRuntimes` attribute contains many types of component Runtime MBeans, one of which is `WebAppComponentRuntimeMBean`. When you get the value of this attribute, you use the child MBean's `Type` attribute to get a specific type of component Runtime MBean.
5. `WebAppComponentRuntimeMBean`, `ServletRuntimes` attribute.

The code in Listing 3-3 navigates the hierarchy described in the previous paragraphs and gets values of `ServletRuntimeMBean` attributes.

Listing 3-3 Monitoring Servlets

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Hashtable;

import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

public class MonitorServlets {
   private static MBeanServerConnection connection;
   private static JMXConnector connector;
   private static final ObjectName service;

   // Initializing the object name for DomainRuntimeServiceMBean
   // so it can be used throughout the class.
   static {
      try {
         service = new ObjectName("weblogic:Name=DomainRuntimeService");
      }catch (MalformedObjectNameException e) {
         throw new AssertionError(e.getMessage());
      }
   }
```

BETA

```
/*
* Initialize connection to the Domain Runtime MBean server
*/
public static void initConnection(String hostname, String portString,
   String username, String password) throws IOException,
   MalformedURLException {
   String protocol = "t3";
   Integer portInteger = Integer.valueOf(portString);
   int port = portInteger.intValue();
   String jndiroot = "/jndi/";
   String mserver = "weblogic.management.mbeanservers.domainruntime";

   JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname,
      port, jndiroot + mserver);
   Hashtable h = new Hashtable();
   h.put(Context.SECURITY_PRINCIPAL, username);
   h.put(Context.SECURITY_CREDENTIALS, password);
   h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
      "weblogic.management.remote");
   connector = JMXConnectorFactory.connect(serviceURL, h);
   connection = connector.getMBeanServerConnection();
}

/*
* Get an array of ServerRuntimeMBeans
*/
public static ObjectName[] getServerRuntimes() throws Exception {
   return (ObjectName[]) connection.getAttribute(service,
      "ServerRuntimes");
}

/*
* Get an array of WebApplicationComponentRuntimeMBeans
*/
public void getServletData() throws Exception {
   ObjectName[] serverRT = getServerRuntimes();
   int length = (int) serverRT.length;
   for (int i = 0; i < length; i++) {
      ObjectName[] appRT =
         (ObjectName[]) connection.getAttribute(serverRT[i],
         "ApplicationRuntimes");
      System.out.println("Application name: " +
         (String)connection.getAttribute(serverRT[i], "Name"));
      int applength = (int) appRT.length;
      for (int x = 0; x < applength; x++) {
         ObjectName[] compRT =
            (ObjectName[]) connection.getAttribute(appRT[x],
            "ComponentRuntimes");
         System.out.println("  Component name: " +
```

BETA

```
                (String)connection.getAttribute(appRT[x], "Name"));
            int complength = (int) compRT.length;
            for (int y = 0; y < complength; y++) {
               String componentType =
                  (String) connection.getAttribute(compRT[y], "Type");
               System.out.println(componentType.toString());
               if (componentType.toString().equals("WebAppComponentRuntime")){
                  ObjectName[] servletRTs = (ObjectName[])
                     connection.getAttribute(compRT[y], "Servlets");
                  int servletlength = (int) servletRTs.length;
                  for (int z = 0; z < servletlength; z++) {
                     System.out.println("    Servlet name: " +
                        (String)connection.getAttribute(servletRTs[z],
                         "Name"));
                     System.out.println("       Servlet context path: " +
                        (String)connection.getAttribute(servletRTs[z],
                         "ContextPath"));
                     System.out.println("       Invocation Total Count : " +
                        (Object)connection.getAttribute(servletRTs[z],
                         "InvocationTotalCount"));
                  }
               }
            }
         }
      }
   }

   public static void main(String[] args) throws Exception {
      String hostname = args[0];
      String portString = args[1];
      String username = args[2];
      String password = args[3];

      MonitorServlets s = new MonitorServlets();
      initConnection(hostname, portString, username, password);
      s.getServletData();
      connector.close();
   }
}
```

BETA

BETA

CHAPTER 4

Managing a Domain's Configuration with JMX

BETA

To integrate third-party management systems with the WebLogic Server management system, WebLogic Server provides standards-based interfaces that are fully compliant with the Java Management Extensions (JMX) specification. Software vendors can use these interfaces to change the configuration of a WebLogic Server domain and monitor the distribution (activation) of those changes to all server instances in the domain. While WebLogic Server requires remote JMX clients to include a small number of WebLogic Server classes on their class path, the JMX client code itself can perform all WebLogic Server management functions without importing proprietary classes. (See "Set Up the Classpath for Remote Clients" on page 3-1.)

To understand the process of changing a WebLogic Server domain and activating the changes, see Managing Configuration Changes in *Understanding Domain Configuration*.

The following sections describe managing a WebLogic Server domain's configuration through JMX:

Editing MBean Attributes: Main Steps

To edit MBean attributes:

1. Start an Edit Session.

   All edits occur within the context of an edit session, and within each WebLogic Server domain, only one edit session can be active at a time. Once a user has started an edit session, WebLogic Server locks other users from accessing the pending Configuration MBean hierarchy. See "Managing Locks" on page 4-17.

2. Change Attributes or Create New MBeans.

   Changing an MBean attribute or creating a new MBean updates the in-memory hierarchy of pending Configuration MBeans. If you end your edit session before saving these changes, the unsaved changes will be discarded.

3. Save Changes to the Pending Configuration Files.

   When you are satisfied with your changes to the in-memory hierarchy, save them to the domain's pending configuration files. Any changes that you save remain in the pending configuration files until they have been activated or explicitly reverted. If you end your edit session before activating the saved changes, you or someone else can activate them in a subsequent edit session.

   You can iteratively make changes and save changes before activating them. For example, you can create and save a server. Then you can configure the new server's listen port and listen address and save those changes. Organizing your code in this way can facilitate correcting nay validation errors.

4. Activate Your Changes.

   When you activate your changes, WebLogic Server copies the saved, pending configuration files to all servers in the domain. Each server evaluates the changes and indicates whether it can consume them. If it can, then it updates its active configuration files and in-memory hierarchy of Configuration MBeans.

For an example of editing MBeans and activating the edits, see "Example: Changing the Administration Port" on page 4-5.

Start an Edit Session

To start an edit session:

1. Initiate a connection to the edit MBean server.

BETA

The connection returns an object of type `java.management.MBeanServerConnection`.

See "Connect to an MBean Server" on page 3-2.

2. Get an object name for `ConfigurationManagerMBean`.

   `ConfigurationManagerMBean` provides methods to start and stop edit sessions, and save, undo, and activate configuration changes. (See ConfigurationManagerMBean in *WebLogic Server MBean Reference*.)

   Each domain has only one instance of `ConfigurationManagerMBean`, and it is contained in the `EditServiceMBean` `ConfigurationManagement` attribute. `EditServiceMBean` is your entry point for all edit operations. It has a simple, fixed object name and contains attributes and operations for accessing all other MBeans in the edit MBean server.

   To get the `ConfigurationManagerMBean` object name, use the following method:

   ```
   MBeanServerConnection.getAttribute(
       ObjectName object-name, String attribute)
   ```

   where:

   – *`object-name`* is the literal `"weblogic:Name=EditService"`, which is the object name of `EditServiceMBean`.

   – *`attribute`* is the literal `"ConfigurationManagement"`, which is the name of the attribute in `EditServiceMBean` that contains `ConfigurationManagerMBean`.

3. Start an edit session.

   To start an edit session, invoke the `ConfigurationManagerMBean` `startEdit(int waitTime, int timeout)` operation where:

   – *`waitTime`* specifies how many milliseconds `ConfigurationManagerMBean` waits to establish a lock on the edit MBean hierarchy. You cannot establish a lock if other edits are in progress unless you have administrator privileges (see "Managing Locks" on page 4-17).

   – *`timeout`* specifies how many milliseconds you have to complete your edit session. If the time expires before you save or activate your edits, all of your unsaved changes are discarded.

   If you are not familiar with using JMX to invoke MBean operations, see "Invoking MBean Operations" on page 4-9.

   The `startEdit` operation returns either of the following:

BETA

- If it cannot establish a lock on the edit tree within the amount of time that you specified, it throws `weblogic.management.mbeanservers.edit.EditTimedOutException`.
- If it successfully locks the edit tree, it returns an object name for `DomainMBean`, which is the root of the edit MBean hierarchy.

Change Attributes or Create New MBeans

To change the attribute values of existing MBeans, create new MBeans, or delete MBeans:

1. Navigate the hierarchy of the edit tree and retrieve an object name for the MBean that you want to edit. To create or delete MBeans, retrieve an object name for the MBean that contains the appropriate factory methods.

   See "Navigate MBean Hierarchies" on page 3-6.

2. To change the value of an MBean attribute, invoke the `MBeanServerConnection.setAttribute(`*`object-name`*`, `*`attribute`*`)` method where:
   - *`object-name`* is the object name of the MBean that you want to edit.
   - *`attribute`* is a `javax.management.Attribute` object, which contains the name of the MBean attribute that you want to change and its new value.

   To create an MBean, invoke the MBean's create method. For example, the factory method to create an instance of `ServerMBean` is `createServer(String` *`name`*`)` in `DomainMBean`.

   In *WebLogic Server MBean Reference*, each MBean describes the location of its factory methods. (See ServerMBean.)

3. (Optional) If you organize your edits into multiple steps, consider validating your changes after each step by invoking the `ConfigurationManagerMBean` `validate()` operation.

   The validate method verifies that all unsaved changes satisfy dependencies between MBean attributes and makes other checks that cannot be made at the time that you set the value of a single attribute.

   If it finds validation errors, the `validate()` operation throws an exception of type `weblogic.management.mbeanservers.edit.ValidationException`. See "Exception Types Thrown by Edit Operations" on page 4-10.

   Validating is optional because the `save()` operation also validates changes before saving.

BETA

Save Changes to the Pending Configuration Files

Save your changes by invoking the `ConfigurationManagerMBean save()` operation.

Activate Your Changes

To activate your saved changes throughout the domain:

1. Invoke the `ConfigurationManagerMBean activate(long timeout)` operation where `timeout` specifies how many milliseconds the operation has to complete.

   The `activate` operation returns an object name for an instance of `ActivationTaskMBean`, which contains information about the activation request. See "Listing and Undoing Changes" on page 4-11.

   When the `activate` operation succeeds or times out, it releases your lock on the editable MBean hierarchy.

2. Close your connection to the MBean server by invoking `JMXConnector.close()`.

Example: Changing the Administration Port

The code example in Listing 4-1 changes the context path that you use to access the Administration Console for a domain. This behavior is defined by the `DomainMBean ConsoleContextPath` attribute.

Note the following about the code example:

- For information on how the class connects to the edit MBean server, see "Connect to an MBean Server" on page 3-2.
- To simplify the code for learning purposes, exception handling in Listing 4-1 is minimal. See "Best Practices: Recommended Pattern for Editing and Handling Exceptions" on page 4-18.

Listing 4-1 Example: Changing the Administration Console's Context Path

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Hashtable;

import javax.management.Attribute;
import javax.management.MBeanServerConnection;
```

BETA

```
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

public class EditWLSMBeans {
   private static MBeanServerConnection connection;
   private static JMXConnector connector;
   private static final ObjectName service;

   // Initializing the object name for EditServiceMBean
   // so it can be used throughout the class.
   static {
      try {
        service = new ObjectName("weblogic:Name=EditService");
      } catch (MalformedObjectNameException e) {
         throw new AssertionError(e.getMessage());
      }
   }

   /**
   * ----------------------------------------------------------
   * Methods to start an edit session.
   * NOTE: Error handling is minimal to help you see the
   *       main steps in editing MBeans. Your code should
   *       include logic to catch and process exceptions.
   * ----------------------------------------------------------
   */

   /*
   * Initialize connection to the edit MBean server.
   */
   public static void initConnection(String hostname, String portString,
      String username, String password) throws IOException,
      MalformedURLException {

      String protocol = "t3";
      Integer portInteger = Integer.valueOf(portString);
      int port = portInteger.intValue();
      String jndiroot = "/jndi/";
      String mserver = "weblogic.management.mbeanservers.edit";

      JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname, port,
      jndiroot + mserver);

      Hashtable h = new Hashtable();
      h.put(Context.SECURITY_PRINCIPAL, username);
      h.put(Context.SECURITY_CREDENTIALS, password);
```

BETA

```
   h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
      "weblogic.management.remote");
      connector = JMXConnectorFactory.connect(serviceURL, h);
      connection = connector.getMBeanServerConnection();
}

/**
* Start an edit session.
*/
public ObjectName startEditSession() throws Exception {
   // Get the object name for ConfigurationManagerMBean.
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(service,
      "ConfigurationManager");

   // Instruct MBeanServerConnection to invoke
   // ConfigurationManager.startEdit(int waitTime int timeout).
   // The startEdit operation returns a handle to DomainMBean, which is
   // the root of the edit hierarchy.
   ObjectName domainConfigRoot = (ObjectName) connection.invoke(cfgMgr,
      "startEdit", new Object[] { new Integer(60000),
      new Integer(120000) }, new String[] { "int", "int" });
   if (domainConfigRoot == null) {
      // Couldn't get the lock
      throw new Exception("Somebody else is editing already");
   }
   return domainConfigRoot;

/**
* ------------------------------------------------------------
* Methods to change MBean attributes.
* ------------------------------------------------------------
*/

/**
* Modify the DomainMBean's ConsoleContextPath attribute.
*/
public void editConsoleContextPath(ObjectName cfgRoot) throws Exception {
   // The calling method passes in the object name for DomainMBean.
   // This method only needs to set the value of an attribute
   // in DomainMBean.
   Attribute adminport = new Attribute("ConsoleContextPath", new String(
      "secureConsoleContext"));
   connection.setAttribute(cfgRoot, adminport);
   System.out.println("Changed the Admin Console context path to " +
      "secureConsoleContext");
}

/**
* ------------------------------------------------------------
* Method to activate edits.
```

BETA

```
   * ------------------------------------------------------------
   */
   public ObjectName activate() throws Exception {
      // Get the object name for ConfigurationManagerMBean.
      ObjectName cfgMgr = (ObjectName) connection.getAttribute(service,
         "ConfigurationManager");
      // Instruct MBeanServerConnection to invoke
      // ConfigurationManager.activate(long timeout).
      // The activate operation returns an ActivationTaskMBean.
      // You can use the ActivationTaskMBean to track the progress
      // of activating changes in the domain.
      ObjectName task = (ObjectName) connection.invoke(cfgMgr, "activate",
         new Object[] { new Long(120000) }, new String[] { "long" });
      return task;
   }

   public static void main(String[] args) throws Exception {
      String hostname = args[0];
      String portString = args[1];
      String username = args[2];
      String password = args[3];

      EditWLSMBeans ewb = new EditWLSMBeans();

      // Initialize a connection with the MBean server.
      initConnection(hostname, portString, username, password);

      // Get an object name for the Configuration Manager.
      ObjectName cfgMgr = (ObjectName) connection.getAttribute(service,
         "ConfigurationManager");

      // Start an edit session.
      ObjectName cfgRoot = ewb.startEditSession();
      // Edit the server log MBeans.
      ewb.editConsoleContextPath(cfgRoot);

      // Save and activate.
      connection.invoke(cfgMgr, "save", null, null);
      ewb.activate();

      // Close the connection with the MBean server.
      connector.close();
   }
}
```

BETA

Invoking MBean Operations

Unlike other J2EE APIs in which you directly invoke a method in an interface (for example, *`interface.method()`*), JMX operations must be invoked indirectly through the `MBeanServerConnection.invoke` operation. This operation takes several parameters which require you to create a few more objects than you would create when invoking other J2EE APIs.

The complete signature of the `invoke` operation is:

```
MBeanServerConnection.invoke(ObjectName name, String operationName,
Object[] params, String[] signature)
```

where:

- `name` is the object name for the MBean that contains the operation
- `operationName` is the name of the operation as defined in the MBean
- `params` is an array of objects. In the array, each object contains the value of a parameter to pass to the MBean operation specified in `name`.
- `signature` is an array of `String` objects that describe the data type of each parameter in the `params` object array.

There are two techniques for constructing the necessary objects and invoking the MBean operation:

- If the MBean operation requires only one or two parameters, you can construct the necessary objects and invoke the operation in two lines. See Listing 4-2.
- If the MBean operation requires several parameters, your code will be easier to read if you use separate lines of code to get the object name for the MBean, construct `Object[]` arrays and `String[]` arrays, and invoke the operation. See Listing 4-3.

For example, to invoke `ConfigurationManagerMBean.startEdit(int waitTime int timeout)` in two lines of code:

1. Get an object name for `ConfigurationManagerMBean`.
2. In the second line, pass the following parameters to the `invoke` method:
   - The object name of `ConfigurationManagerMBean`
   - The literal `"startEdit"`
   - The constructor for an `Object[]` that contains the parameter values.
   - The constructor for a `String[]` that describes the data type of each parameter value.

BETA

Listing 4-2 Example: Invoking an MBean Operation in Two Lines

```
ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
   "ConfigurationManager");

connection.invoke(cfgMgr, "startEdit",
    new Object[]{new Integer(60000), new Integer(120000)},
    new String[]{"int", "int"}
   );
```

Listing 4-3 Example: Invoking an MBean Operation that Requires Several Parameters

```
ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
   "ConfigurationManager");

Object[] params = new Object[]{new Integer(6000), new Integer(120000)};
String[] paramTypes = new String[]{"int", "int"};

connection.invoke(cfgMgr, "startEdit", params, paramTypes);
```

BETA

Exception Types Thrown by Edit Operations

Table 4-1 describes all of the exception types that WebLogic Server can throw during edit operations. When WebLogic Server throws such an exception, the MBean server wraps the exception in `javax.management.MBeanException`. (See the J2SE Javadoc for `MBeanException`.)

Table 4-1 Exception Types Thrown by Edit Operations

Exception Type	Thrown When
`EditTimedOutException`	The request to start an edit session times out.
`NotEditorException`	You attempt to edit MBeans without having a lock or when an administrative user cancels your lock and starts an edit session.
`ValidationException`	You set an MBean attribute's value to the wrong data type, outside an allowed range, not one of a specified set of values, or incompatible with dependencies in other attributes.

Listing and Undoing Changes

The following sections describe working with changes that you have made during an edit session:

WebLogic Server describes changes in a serializable object of type `weblogic.management.mbeanservers.edit.Change`.

Note: As of this Beta release, there is no public interface for `Change` objects. To view information about a change, invoke `Change.toString()`. In the final release, `Change` objects will be of type `javax.management.openmbean.TabularData`.

Through JMX, you can access information about the changes to a domain's configuration that have occurred during the current server session only. WebLogic Server maintains an archive of configuration files, but the archived data and comparisons of archive versions is not available through JMX.

List Unsaved Changes

For each change that you make to an MBean attribute, WebLogic Server creates a `Change` object which contains information about the change. You can access these objects from the `ConfigurationManagerMBean` `Changes` attribute until you save the changes. See ConfigurationManagerMBean.Changes in *WebLogic Server MBean Reference*.

Any unsaved changes are discarded when your edit session ends.

To list unsaved changes:

1. Start an edit session and change at least one MBean attribute.
2. Get the value of the `ConfigurationManagerMBean` `Changes` attribute and assign the output to a variable of type `Object[]`.
3. For each object in the array, invoke `Object.toString()` to output a description of the change.

BETA

Note: As of this Beta release, there is no public interface for `Change` objects. To view information about a change, invoke `Change.toString()`. In the final release, `Change` objects will be of type `javax.management.openmbean.TabularData`.

The code in Listing 4-4 creates a method that lists unsaved changes. It assumes that the calling method has already established a connection to the edit MBean server.

Listing 4-4 Example Method that Lists Unsaved Changes

```
public void listUnsaved() throws Exception {
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
   "ConfigurationManager");
   Object[] list = (Object[])connection.getAttribute(cfgMgr, "Changes");
   int length = (int) list.length;
   for (int i = 0; i < length; i++) {
      System.out.println("Unsaved change: " + list[i].toString());
   }
}
```

BETA

List Unactivated Changes

When anyone saves changes, WebLogic Server persists the changes in the pending configuration files. The changes remain in these files, even across multiple editing sessions, unless someone invokes the `ConfigurationManagerMBean undoUnactivatedChanges()` operation, which reverts all unactivated changes from the pending files.

The `ConfigurationManagerMBean UnactivatedChanges` attribute contains `Change` objects for both unsaved changes and changes that have been saved but not activated. (There is no attribute that contains only saved but unactivated changes.) See ConfigurationManagerMBean.UnactivatedChanges in *WebLogic Server MBean Reference*.

To list changes that you have saved in the current editing session but not activated, or changes that your or others have saved in previous editing sessions but not activated:

1. Start an edit session and change at least one MBean attribute.
2. Get the value of the `ConfigurationManagerMBean UnactivatedChanges` attribute and assign the output to a variable of type `Object[]`.
3. For each object in the array, invoke `Object.toString()` to output a description of the change.

Note: As of this Beta release, there is no public interface for `Change` objects. To view information about a change, invoke `Change.toString()`. In the final release, `Change` objects will be of type `javax.management.openmbean.TabularData`.

The code in Listing 4-5 creates a method that lists unactivated changes. It assumes that the calling method has already established a connection to the edit MBean server.

Listing 4-5 Example Method that Lists Unactivated Changes

```
public void listUnactivated() throws Exception {
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
   "ConfigurationManager");
   Object[] list = (Object[])connection.getAttribute(cfgMgr,
      "UnactivatedChanges");
   int length = (int) list.length;
   for (int i = 0; i < length; i++) {
      System.out.println("Unactivated changes: " + list[i].toString());
   }
}
```

List Changes in the Current Activation Task

When you activate changes, WebLogic Server creates an instance of `ActivationTaskMBean`, which maintains the list of changes that it activated. You can access these `ActivationTaskMBean`s from either of the following:

- The `ConfigurationManagerMBean` `activate()` method returns an object name for the `ActivationTaskMBean` that describes the current activation task.
- The `ConfigurationManagerMBean` `CompletedActivationTasks` attribute can potentially contain a list of all `ActivationTaskMBean` instances that have been created during the current Administration Server instantiation. See "Listing All Activation Tasks Stored in Memory" on page 4-16.

To list changes in the current activation task only:

1. Start an edit session.
2. Assign the output of the `activate` operation to an instance variable of type `javax.management.ObjectName`.

BETA

3. Get the value of the `Changes` attribute. Invoke `Object.toString()` to output the value of the `Change` object.

   Note: As of this Beta release, there is no public interface for `Change` objects. To view information about a change, invoke `Change.toString()`. In the final release, `Change` objects will be of type `javax.management.openmbean.TabularData`.

The code in Listing 4-6 creates a method that lists all changes activated in the current editing session. It assumes that the calling method has already established a connection to the edit MBean server.

Listing 4-6 Example Method that Lists Changes in the Current Activation Task

```
public void activateAndList()
   throws Exception {
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
      "ConfigurationManager");
   ObjectName task = (ObjectName) connection.invoke(cfgMgr, "activate",
      new Object[] { new Long(120000) }, new String[] { "long" });
   Object[] changes = (Object[])connection.getAttribute(task, "Changes");
   int i = (int) changes.length;
   for (int i = 0; i< i; i++) {
      System.out.println("Changes activated: " + changes[i].toString());
   }
}
```

BETA

Undoing Changes

`ConfigurationManagerMBean` provides two operations for undoing changes made during an editing session:

- `undo`

  Reverts unsaved changes.

- `undoUnactivatedChanges`

  Reverts all changes, saved or unsaved, that have not yet been activated. If other users have saved changes in a previous editing session but not activated those changes, invoking the `ConfigurationManagerMBean` `undoUnactivatedChanges()` operation reverts those changes as well.

After you invoke this method, the pending configuration files are identical to the working configuration files that the active servers use.

To undo changes, start an edit session and invoke the `ConfigurationManagerMBean` `undo` or `undoUnactivatedChanges` operation.

For example:

```
connection.invoke(cfgMgr, "undo", null, null);
```

Tracking the Activation of Changes

In addition to maintaining a list of changes, each `ActivationTaskMBean` that WebLogic Server creates when you invoke the `activate` operation describes which user activated the changes, the status of the activation task, and the time at which the changes were activated.

The Administration Server maintains instances of `ActivationTaskMBean` in memory only; they are not persisted and are destroyed when you shut down the Administration Server. Because the `ActivationTaskMBean` instances contain a list of `Change` objects (each of which describes a single change to an MBean attribute), they use a significant amount of memory. To save memory, by default the Administration Server maintains only a few of the most recent `ActivationTaskMBean` instances in memory. To change the default, increase the value of the `ConfigurationManagerMBean` `CompletedActivationTasksCount` attribute.

The following sections describe working with instances of `ActivationTaskMBean`:

- "Listing the Status of the Current Activation Task" on page 4-15
- "Listing All Activation Tasks Stored in Memory" on page 4-16
- "Purging Completed Activation Tasks from Memory" on page 4-17

Listing the Status of the Current Activation Task

When you invoke the `activate` operation, WebLogic Server returns an `ActivationTaskMBean` instance to represent the activation task.

The `ActivationTaskMBean` `State` attribute describes the status of the activation task. This attribute stores an `int` value and `ActivationTaskMBean` defines constants for each of the `int` values. See ActivationTaskMBean in *WebLogic Server MBean Reference*.

To list the status of the current activation task:

1. Start an edit session and change at least one MBean attribute.

BETA

2. Invoke the `ConfigurationManagerMBean` `activate(long timeout)` operation and assign the output to a variable of type `ActivationTaskMBean`.
3. Get the value of the `ActivationTaskMBean` `State` attribute.

Listing All Activation Tasks Stored in Memory

The `ActivationTaskMBean` that the `activate` operation returns describes only a single activation task. The Administration Server keeps this `ActivationTaskMBean` in memory until you purge it or the number of activation tasks exceeds the value of the `ConfigurationManagerMBean` `CompletedActivationTasksCount` attribute.

To access all `ActivationTaskMBean` instances that are currently stored in memory:

1. Connect to the edit MBean server. (You do not need to start an edit session.)
2. Get the value of the `ConfigurationManagerMBean` `CompletedActivationTasks` attribute and assign the output to a variable of type `Object[]`.
3. (Optional) For each object in the array, get and print the value of `ActivationTaskMBean` attributes such as `User` and `State`.

   See ActivationTaskMBean in *WebLogic Server MBean Reference*.
4. (Optional) For each object in the array, get the value of the `Changes` attribute. Invoke `Object.toString()` to output the value of the `Change` object.

Listing 4-7 Example Method that Lists All Activation Tasks in Memory

```
public void listActivated() throws Exception {
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(SERVICE,
   "ConfigurationManager");
   ObjectName[] list = (ObjectName[])connection.getAttribute(cfgMgr,
      "CompletedActivationTasks");
   System.out.println("Listing completed activation tasks.");
   int length = (int) list.length;
   for (int i = 0; i < length; i++) {
      System.out.println("Activation task " + i);
      System.out.println("User who started activation: " +
         connection.getAttribute(list[i], "User"));
      System.out.println("Task state: " + connection.getAttribute(list[i],
         "State"));
      System.out.println("Start time: " + connection.getAttribute(list[i],
         "StartTime"));
```

BETA

```
      Object[] changes = (Object[])connection.getAttribute(list[i], "Changes");
      int l = (int) changes.length;
      for (int y = 0; y < l; y++) {
         System.out.println("Changes activated: " + changes[y].toString());
      }
   }
}
```

Purging Completed Activation Tasks from Memory

Because the `ActivationTaskMBean` instances contain a list of `Change` objects (each of which describes a single change to an MBean attribute), they use a significant amount of memory.

If your the Administration Server is running out of memory, you can purge completed activation tasks from memory. Then decrease the value of the `ConfigurationManagerMBean CompletedActivationTasksCount` attribute.

To purge completed activation tasks from memory, connect to the edit MBean server and invoke the `ConfigurationManagerMBean purgeCompletedActivationTasks` operation.

For example:

```
connection.invoke(cfgMgr, "purgeCompletedActivationTasks", null, null);
```

Managing Locks

To prevent changes that could leave the pending Configuration MBean hierarchy in an inconsistent state, only one user at a time can edit MBeans. When a user invokes the `ConfigurationManagerMBean startEdit` operation, the `ConfigurationManagerMBean` prevents other users (locks) from starting edit sessions.

The following actions remove the lock:

- The `ConfigurationManagerMBean activate` operation succeeds or times out.

  You can use the `ActivationTaskMBean waitForTaskCompletion` operation to block until the activation process is complete.

- The `ConfigurationManagerMBean stopEdit` operation succeeds.

- A user with administrator privileges invokes the `ConfigurationManagerMBean cancelEdit` operation while another user has the lock.

  For example, `connection.invoke(cfgMgr, "cancelEdit", null, null);`

BETA

All unsaved changes are lost when the lock is removed.

The `ConfigurationManagerMBean` does not prevent multiple users start an edit session under the same, administrative user identity. In such a case, each user is allowed to edit MBeans and save changes. When any of the users activates changes, all changes that have been saved are activated.

Best Practices: Recommended Pattern for Editing and Handling Exceptions

BEA recommends that you organize your editing code into several try-catch blocks. Such an organization will enable you to catch specific types of errors and respond appropriately. For example, instead of abandoning the entire edit session if a change is invalid, your code can save the changes, throw an exception and exit without attempting to activate invalid changes.

Consider using the following structure (see the pseudo-code in Listing 4-8):

- A try block that connects to the edit MBean server, starts an edit session, and makes and saves changes.

  After this try block, one catch block for each of the following types of exceptions:

  – `EditTimedOutException`

    This exception is thrown if the `ConfigurationManagerMBean startEdit()` operation cannot get a lock within the amount of time that you specify.

  – `NotEditorException`

    This exception is thrown if the edit session times out or an administrator cancels your edit session. (See "Managing Locks" on page 4-17.)

  – `ValidationException`

    This exception is thrown if you set a value in an MBean that is the wrong data type, outside an allowed range, not one of a specified set of values, or incompatible with dependencies in other attributes.

    Within this `ValidationException` catch block, include another try block that either attempts to correct the validation error or stops the edit session by invoking the `ConfigurationManagerMBean stopEdit()` operation. If the try block stops the edit session, its catch block should ignore the `NotEditorException`. This exception indicates that you no longer have a lock on the pending Configuration MBean hierarchy; however, because you want to abandon changes and release your lock anyway, it is not an error condition for this exception to be thrown.

BETA

- A try block that activates the changes that have been saved.

  The `ConfigurationManager activate(long timeout)` operation returns an instance of `ActivationTaskMBean`, which contains information about the activation task. BEA recommends that you set the timeout period for `activate()` to a minute and then check the value of the `ActivationTaskMBean State` attribute.

  If `State` contains the constant `STATE_COMMITTED`, then your changes have been successfully activated in the domain. You can use a `return` statement at this point to end your editing work. The lock that you created with `startEdit()` releases after the activation task succeeds.

  If `State` contains a different value, the activation has not succeeded in the timeout period that you specified in `activate(long timeout)`. You can get the value of the `ActivationTaskMBean Error` attribute to find out why.

  After this try block, one catch block to catch the following type of exception:

  – `NotEditorException`

    If this exception is thrown while trying to activate changes, your changes were not activated because your edit session timed out or was cancelled by an administrator.

- (Optional) A try block that undoes the saved changes.

  If your class does not return in the activation try block, then your activation task was not successful. If you do not want these saved changes to be activated by a future attempt to activate changes, then invoke the `ConfigurationManagerMBean undoUnactivatedChanges()` operation.

  Otherwise, the pending configuration files retain your saved changes. The next time any user attempts to activate saved changes, WebLogic Server will attempt to activate your saved changes along with any other saved changes.

  After this try block, one catch block to **ignore** the following type of exception:

  – `NotEditorException`

- A try block to stop the edit session.

  If your activation attempt fails and you are ready to abandon changes, there is no need to wait until your original timeout period to expire. You can stop editing immediately.

  After this try block, one catch block to **ignore** the following type of exception:

  – `NotEditorException`

- Throw the exception that is stored in the `ActivationTaskMBean Error` attribute.

BETA

Listing 4-8 Pseudo-Code Outline for Editing and Exception Handling

```
try {
   //Initialize the connection and start the edit session
...
   ObjectName domainConfigRoot = (ObjectName) connection.invoke(cfgMgr,
      "startEdit",
      new Object[] { new Integer(30000), new Integer(300000) },
      new String[] { "int", "int" });

   // Modify the domain
   ...
   // Save your changes
    connection.invoke(cfgMgr, "save", null, null);

} catch (EditTimedOutException e) {
   // Could not get the lock. Notify user
   ...
   throw new MyAppCouldNotStartEditException(e);
} catch (NotEditorException e) {
  ...
   throw new MyAppEditSessionFailed(e);
} catch (ValidationException e) {
   ...
   try {
      connection.invoke(cfgMgr, "stopEdit", null, null);
   // A NotEditorException here indicates that you no longer have a
   // lock on the pending Configuration MBean hierarchy; however,
   // because you want to abandon changes and release your lock anyway,
   // it is not an error condition for this exception to be thrown
   // and you can safely ignore it.
   } catch (NotEditorException ignore) {
   }
   ...
   throw new MyAppEditChangesInvalid(e);
}

// Changes have been saved, now activate them
try {
   // Activate the changes
   ActivationTaskMBean task = (ObjectName) connection.invoke(cfgMgr,
      "activate",
      new Object[] { new Long(60000) },
      new String[] { "long" });

   // Everything worked, just return.
   String status = (String) connection.getAttribute(task, "State");
   if (status.equals("4"))
   return;
```

BETA

```
   // If there is an activation error, use ActivationTaskMBean.getError
   // to get information about the error
   failure = connection.getAttribute(task, "Error");

// If you catch NotEditorException, your changes were not activated because
// your edit session ended or was cancelled by an administrator. Throw the
// exception
} catch (NotEditorException e) {
   ...
   throw new MyAppEditSessionFailed(e);
}

// If your class executes the remaining lines, it is because activating you
// saved changes failed.

// Optional: You can undo the saved changes that failed to activate. If you
// do not undo your saved changes, they will be activated the next time
// someone attempts to activate changes.
// try {
//     {
      connection.invoke(cfgMgr, "undoUnactivatedChanges", null, null);
// } catch (NotEditorException e) {
//    ...
//    throw new MyAppEditSessionFailed(e);
// }

// Stop the edit session
try {
   connection.invoke(cfgMgr, "stopEdit", null, null);
   // If your activation attempt fails and you are ready to abandon
   // changes, there is no need to wait until your original timeout
   // period to expire. You can stop editing immediately
   // and you can safely ignore any NotEditorException.
   } catch (NotEditorException ignore) {
}
...
// Output the information about the error that caused the activation to
// fail.
throw new MyAppEditSessionFailed(connection.getAttribute(task, "Error"));
```

BETA

BETA

Index

BETA