BEA WebLogic Server™

Programming WebLogic Management Services with JMX

Release 8.1
Revised: October 8, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

1. Overview of WebLogic JMX Services

2. Accessing WebLogic Server MBeans

3. WebLogic Server Management Namespace

4. Accessing and Changing Configuration Information

5. Accessing Runtime Information

6. Using WebLogic Server MBean Notifications and Monitors

7. Using the WebLogic Timer Service to Generate and Receive Notifications

About This Document

This document describes how to use the BEA WebLogic Server™ management APIs to configure and monitor WebLogic Server domains, clusters, and server instances.

The document is organized as follows:

- Chapter 1, "Overview of WebLogic JMX Services" describes the WebLogic Server management interface and provides overviews of WebLogic Server MBeans, MBean home interfaces, and the distributed management architecture.
- Chapter 2, "Accessing WebLogic Server MBeans," describes how to access interfaces for working with WebLogic Server MBeans.
- Chapter 4, "Accessing and Changing Configuration Information," provides examples of retrieving and modifying the configuration of WebLogic Server resources.
- Chapter 5, "Accessing Runtime Information," provides examples of retrieving and modifying runtime information about WebLogic Server domains and server instances.
- Chapter 6, "Using WebLogic Server MBean Notifications and Monitors," describes how to observe and respond to changes in the values of WebLogic Server MBean attributes.
- Chapter 7, "Using the WebLogic Timer Service to Generate and Receive Notifications," which describes configuring the timer service to emit notifications at specific dates and times or at a constant interval.

Note: The WebLogic Security Service provides MBeans and tools for generating additional MBeans that manage security on a WebLogic Server. These MBeans are called Security MBeans and their usage model is different from the one described in this document. For information on Security MBeans, refer to *Developing Security Providers for WebLogic Server*.

Audience

This document is written for independent software vendors (ISVs) and other developers who are interested in creating custom applications that use BEA WebLogic Server facilities to monitor and configure applications and server instances. It assumes that you are familiar with the BEA WebLogic Server platform and the Java programming language, but not necessarily with Java Management Extensions (JMX).

While the document describes how to access and use the Managed Beans (MBeans) that WebLogic Server provides, it does not describe how to create your own, additional MBeans. For information about creating and using MBeans in addition to the ones that WebLogic Server provides, refer to the JMX 1.0 specification, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html

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.

`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;`

Convention	Usage

UPPERCASE MONOSPACE TEXT	Device names, environment variables, and logical operators.
	*Example*s:
	`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}`
. . .	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.

CHAPTER 1

Overview of WebLogic JMX Services

WebLogic Server implements the Sun Microsystems, Inc. Java Management Extensions (JMX) 1.0 specification to provide open and extensible management services. WebLogic Server adds its own set of convenience methods and other extensions to facilitate working in the WebLogic Server distributed environment.

All WebLogic Server resources are managed through these JMX-based services, and third-party services and applications that run within WebLogic Server can be managed through them as well. You can build your own management utilities that use these JMX services to manage WebLogic Server resources and applications.

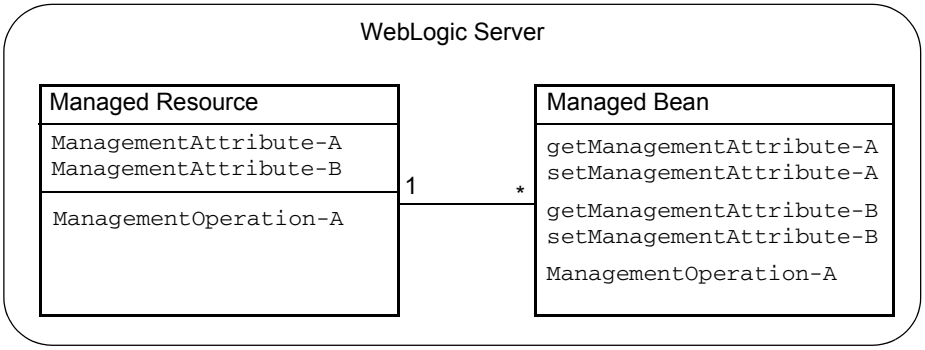
The following sections provide an overview of the WebLogic Server JMX services:

To view the JMX 1.0 specification, download it from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The API documentation is included in the archive that you download.

WebLogic Server Managed Resources and MBeans

Subsystems within WebLogic Server (such as JMS Provider and JDBC Container) and the items that they control (such as JMS servers and JDBC connection pools) are called **WebLogic Server managed resources**. Each managed resource includes a set of attributes that can be configured and monitored for management purposes. For example, each JDBC connection pool includes attributes that define its name, the name of its driver, its initial capacity, and its cache size. Some managed resources provide additional methods (operations) that can be used for management purposes. The WebLogic JMX services expose these management attributes and operations through one or more managed beans (MBeans). An **MBean** is a concrete Java class that is developed per JMX specifications. It can provide getter and setter operations for each management attribute within a managed resource along with additional management operations that the resource makes available. (See Figure 1-1.)

Figure 1-1 Managed Resources and Managed Beans



WebLogic Server MBeans that expose attributes and operations for configuring a managed resource are called **Configuration MBeans** while MBeans that provide information about the runtime state of a managed resource are called **Runtime MBeans**. The functions of configuring resources and viewing data about the runtime state of resources in a WebLogic Server domain are different enough that Configuration MBeans and Runtime MBeans are distributed and maintained differently.

The following sections describe how WebLogic Server distributes and maintains 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 applications, EJBs, and other resources developed onto the Managed Servers and use the Administration Server for configuration and management purposes only.

Note: WebLogic Server does not support multi-domain interaction using either the Administration Console, the `weblogic.Admin` utility, or WebLogic Ant tasks. This restriction does not, however, explicitly preclude a user written Java application from accessing multiple domains simultaneously.

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 "Overview of WebLogic Server Domains" in *Configuring and Managing WebLogic Server*.

MBeans for Configuring Managed Resources

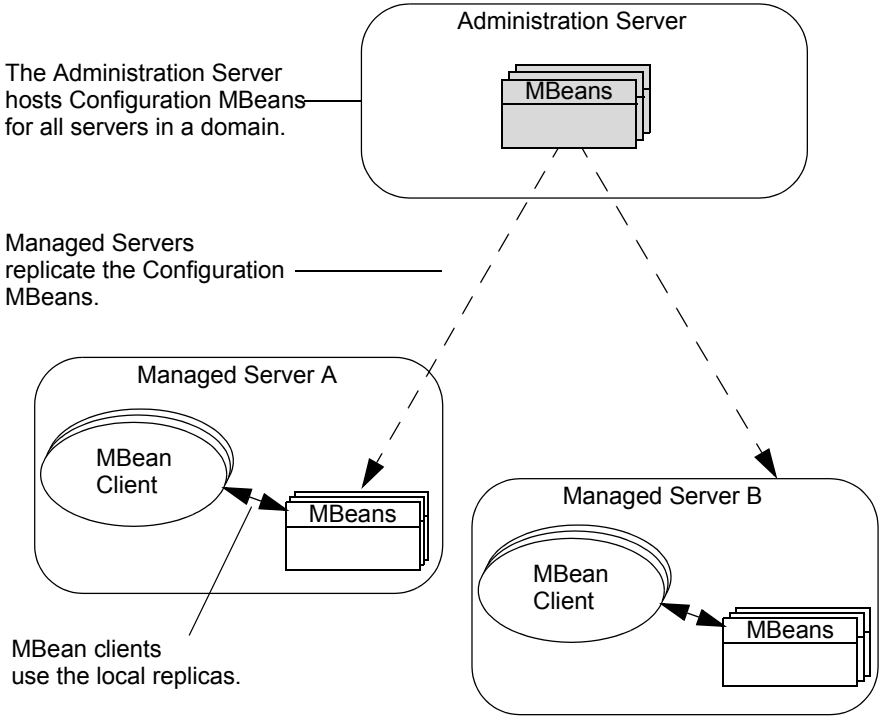
To support the WebLogic Server model of centralizing management responsibilities on the Administration Server, the Administration Server hosts Configuration MBeans for all managed resources on all server instances in the domain. In addition, the Administration Server saves changes to configuration data so that it is available when you shut down and restart a server instance.

To change the configuration of a WebLogic Server resource, you modify the values in the Configuration MBeans on the Administration Server.

Local Replicas of Configuration MBeans

To enhance performance, each Managed Server creates local replicas of all Configuration MBeans in a domain. WebLogic Server subsystems and applications that interact with MBeans use the replicas on the local server instead of initiating remote calls to the Administration Server. (See Figure 1-2.)

Figure 1-2 MBean Replication



The Configuration MBeans on the Administration Server are called **Administration MBeans**, and the replicas on the Managed Servers are called **Local Configuration MBeans**.

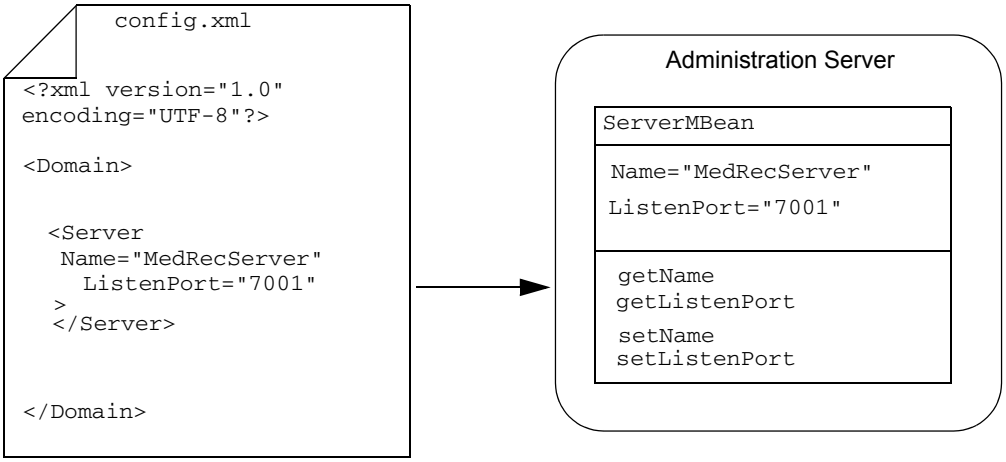
Note: In addition to hosting Administration MBeans, the Administration Server hosts the Local Configuration MBeans that are used by its own subsystems and by any applications that are deployed on the Administration Server.

The Life Cycle of Configuration MBeans

This section describes how Administration MBeans and Local Configuration MBeans are initialized, how changes to configuration data are propagated throughout the WebLogic Server system, and how attribute values can be changed so that they are available when you restart server instances:

1. The life cycle of a Configuration MBean begins when you start the Administration Server. During its startup cycle, the Administration Server initializes all the Administration MBeans for the domain with data from the domain's `config.xml` file. (See Figure 1-3.)

Figure 1-3 Initializing Configuration MBeans

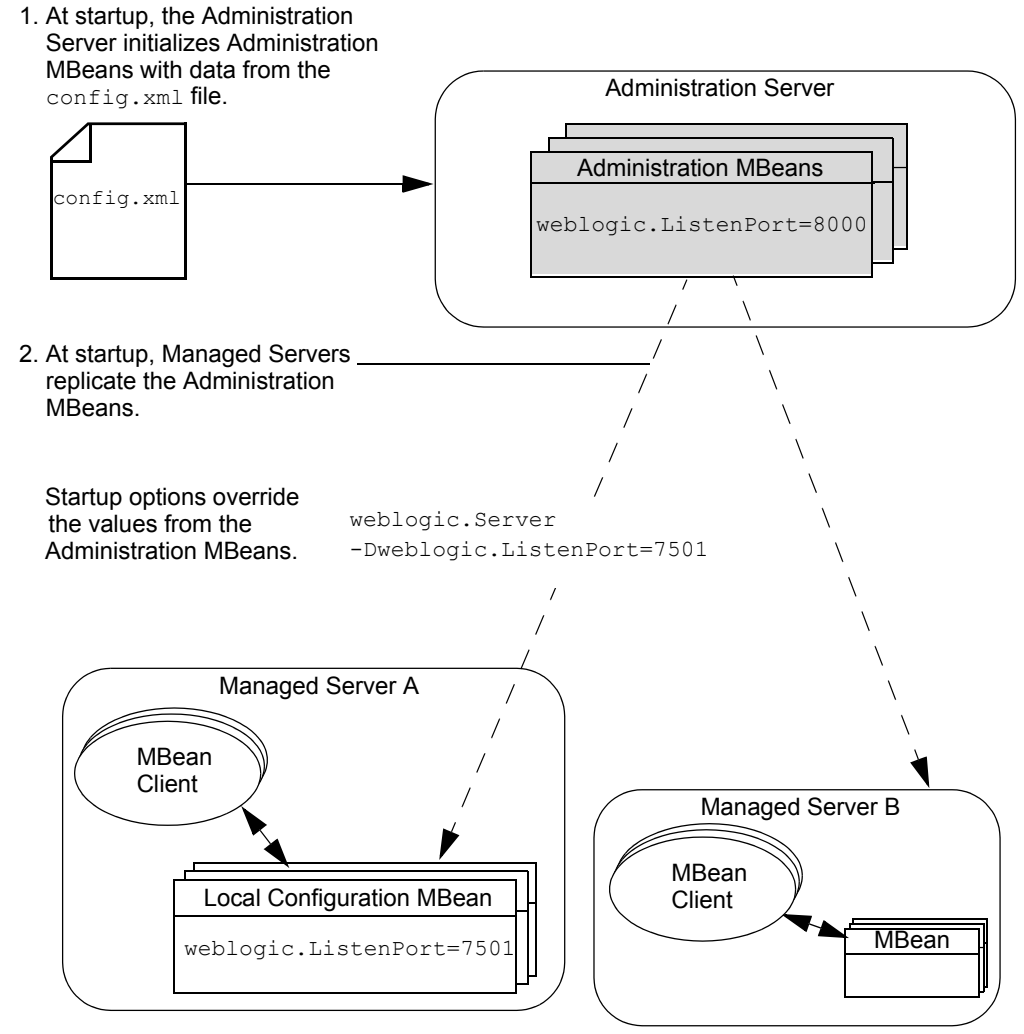


   The Administration Server reads data from the `config.xml` file only during its startup cycle.

2. When a Managed Server starts, it contacts the Administration Server for its configuration data. By default, it creates replicas of the Administration MBeans that configure resources in the domain. However, you can use arguments in the server's startup command to override values of the Administration MBeans.

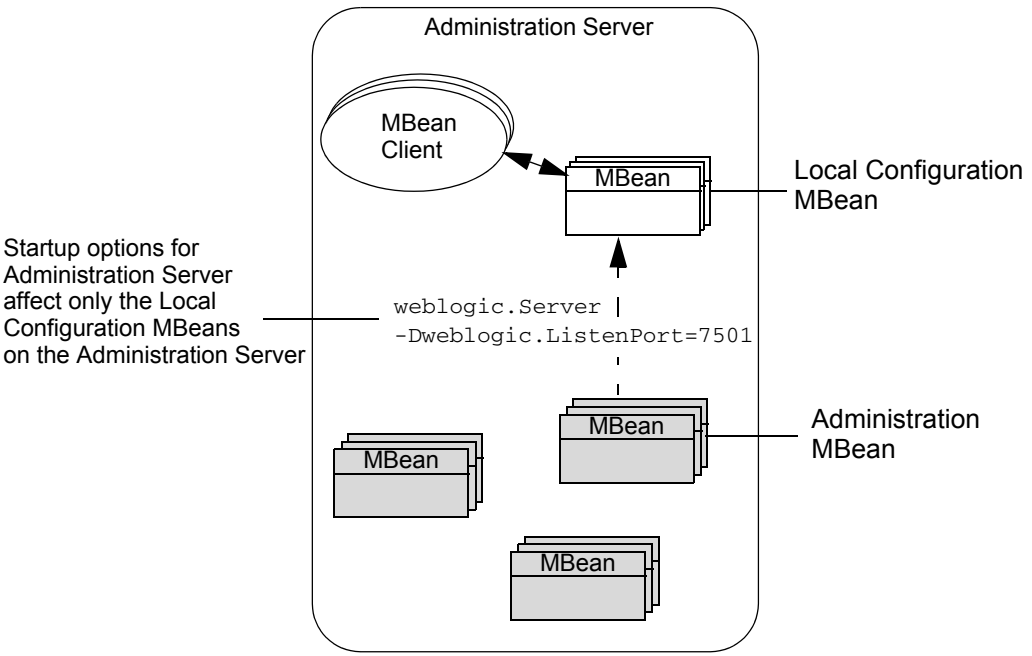
   For example, for Managed Server A, the `config.xml` file states that its listen port is 8000. When you use the `weblogic.Server` command to start Managed Server A, you include the `-Dweblogic.ListenPort=7501` startup option to change the listen port for the current server session. The Managed Server creates a replica of the Administration MBeans, but substitutes `7501` as the value of its listen port. When you restart Managed Server A, it will revert to using the value from the `config.xml` file, `8000`. (See Figure 1-4.)

Figure 1-4 Overriding Administration MBean Values



When you start an Administration Server, any startup command arguments that you use to override the values in `config.xml` affect only the values of the Local Configuration MBeans on the Administration Server. The command arguments do not affect the values of the Administration MBeans and therefore do not affect subsequent server sessions. (See Figure 1-5.)

Figure 1-5 Overriding Values on the Administration Server



3. If you change a value in an Administration MBean, and if the corresponding Managed Server is running, the Administration Server propagates the change to the Local Configuration MBean. Depending on the attribute, the underlying resource might not be able to accept the new value until it restarts. The WebLogic Server Javadoc indicates whether a managed resource can accept new values for an attribute during the current session. Even if a managed resource can accept new values, depending on the frequency with which the resource checks for configuration changes, the resource might not use the updated value immediately.

   Note: BEA recommends that you change only the values of Administration MBean attributes. Do not change attribute values in Local Configuration MBeans. When the Managed Server replicates the data of other Managed Servers, it uses the values that are stored in Administration MBeans. Communication problems can occur if the values in Administration MBeans and Local Configuration MBeans differ.

4. Periodically, the Administration Server determines whether Administration MBeans have been changed and writes any changes back to `config.xml`. Changes also are written to `config.xml` when the Administration Server shuts down or when MBean attributes are modified by a WebLogic Server utility such as the Administration Console or `weblogic.Admin`.

5. Local Configuration MBeans are destroyed when you shut down Managed Servers. Administration MBeans are destroyed when you shut down the Administration Server.

Replication of MBeans for 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 its Local Configuration MBeans, it also contains a copy of all Administration MBeans for the domain.

Do not interact with these Administration MBeans on a Managed Server. They reflect the last known configuration for the domain and are used only for starting the Managed Server in MSI mode. Modifying an Administration MBean on a Managed Server can cause the Managed Server's configuration to be inconsistent with the Administration Server, which will lead to unpredictable results. In addition, Managed Servers are not aware of the Administration MBeans on other Managed Servers.

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

Documentation for Configuration MBean APIs

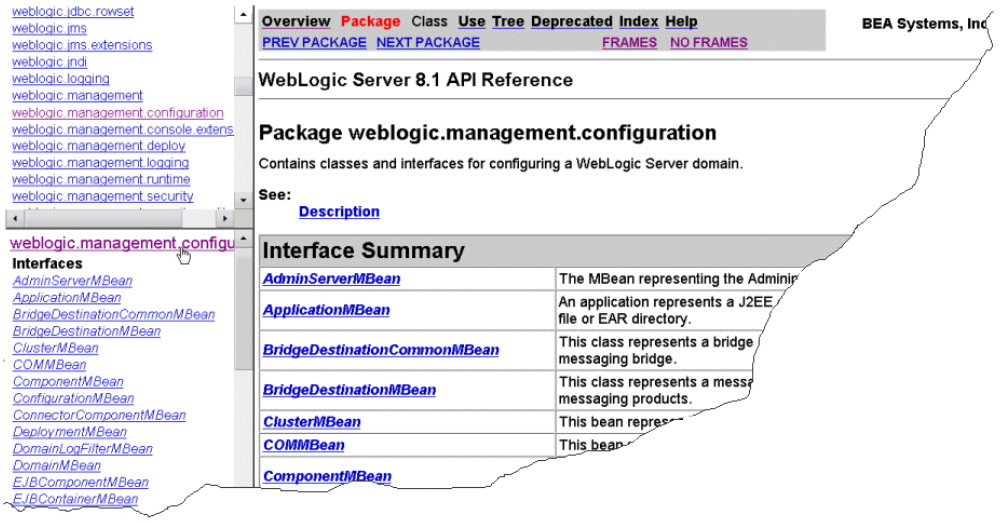
To view the documentation for Configuration MBeans:

1. Open the WebLogic Server Javadoc.
2. In the top left pane of the Web browser, click `weblogic.management.configuration`.

   The lower left pane displays links for the package.
3. In the lower left pane, click `weblogic.management.configuration` again.

   The right pane displays the package summary. (See Figure 1-6.)

Figure 1-6 Javadoc for the configuration Package



4. Click on an interface name to view its API documentation.

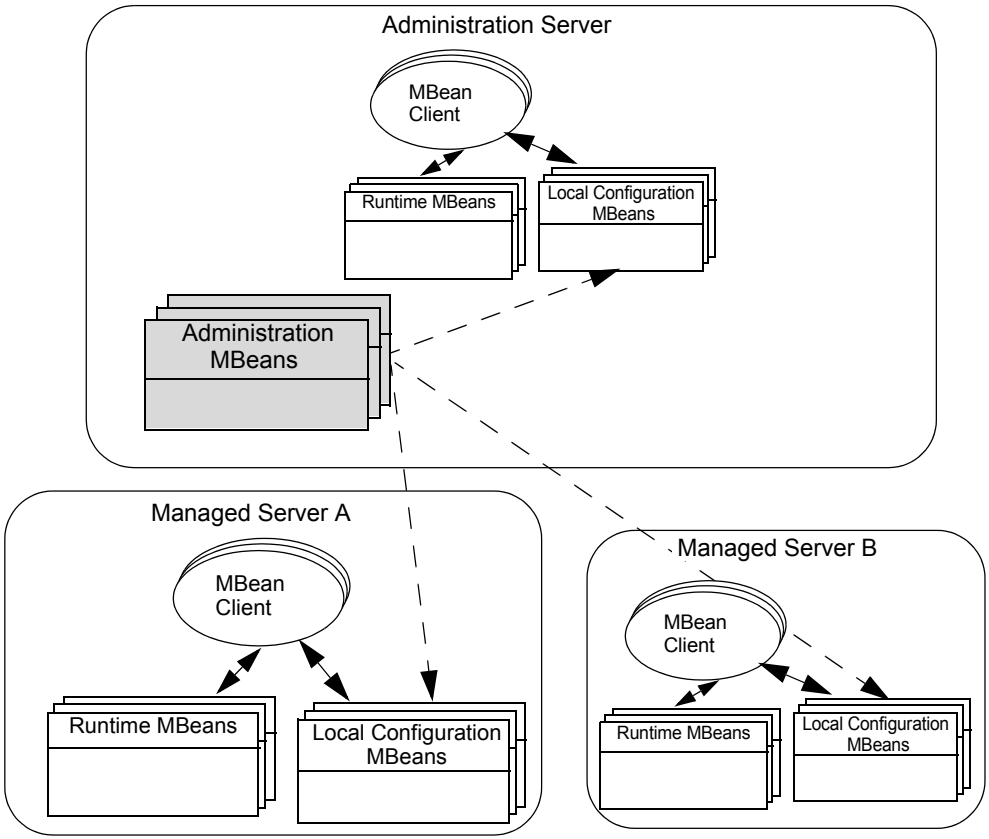
MBeans for Viewing the Runtime State of Managed Resources

WebLogic Server managed resources provide performance metrics and other information about their runtime state through one or more Runtime MBeans. Runtime MBeans are not replicated like Configuration MBeans, and they exist only on the same server instance as their underlying managed resources.

Because Runtime MBeans contain only transient data, they do not save their data in the `config.xml` file. When you shut down a server instance, all runtime statistics and metrics from the Runtime MBeans are destroyed.

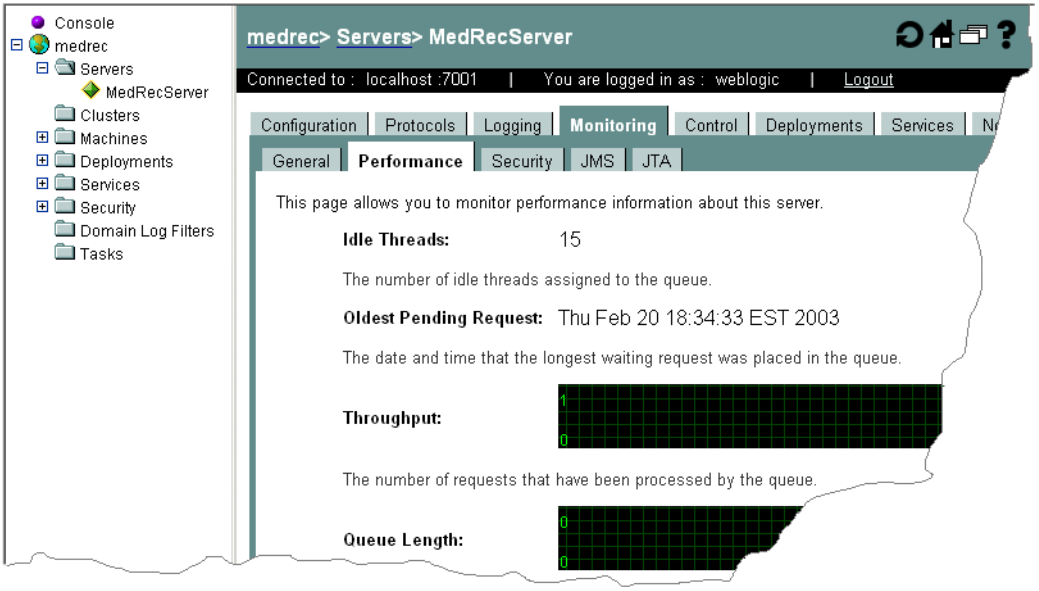
The following figure (Figure 1-7) illustrates how Runtime MBeans, Administration MBeans, and Local Configuration MBeans are distributed throughout a domain.

Figure 1-7 Distribution of MBeans



You can use the Administration Console, the `weblogic.Admin` utility, or MBean APIs to view the values. (See Figure 1-8.)

Figure 1-8 Viewing Runtime Metrics from the Administration Console



You can also use these interfaces to change some runtime values. For example, the `weblogic.management.runtime.DeployerRuntimeMBean` activates and deactivates a deployed module by changing its runtime state.

Documentation for Runtime MBean APIs

To view the documentation for Runtime MBeans:

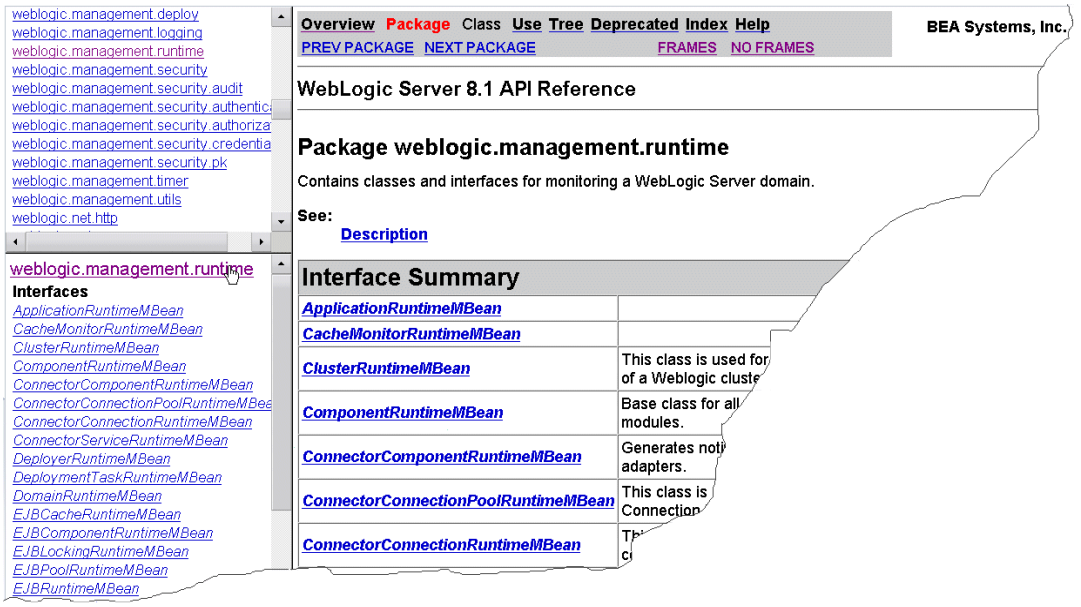
1. Open the WebLogic Server Javadoc.
2. In the top left pane of the Web browser, click `weblogic.management.runtime`.

   The lower left pane displays links for the package.

3. In the lower left pane, click `weblogic.management.runtime` again.

   The right pane displays the package summary. (See Figure 1-9.)

Figure 1-9 Javadoc for the runtime Package



4. Click on an interface name to view its API documentation.

Security MBeans

The WebLogic Security Service provides MBeans and tools for generating additional MBeans that manage security on a WebLogic Server. These MBeans are called Security MBeans and their usage model is different from the one described in this document. For information on Security MBeans, refer to *Developing Security Providers for WebLogic Server*.

Non-WebLogic Server MBeans

WebLogic Server provides hundreds of MBeans, many of which are used to configure and monitor EJBs, Web applications, and other deployable J2EE modules. If you want to use additional MBeans to configure your applications or services, you can create your own MBeans.

Any MBeans that you create can take advantage of the full set of JMX 1.0 features, as defined by the JMX specification (which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html).

However, only MBeans that are provided by WebLogic Server can use the WebLogic Server extensions to JMX. For example, any MBeans that you create for your applications cannot save data in the `config.xml` file and they cannot use the type-safe interface as described in the next section, "MBean Servers and the MBeanHome Interface."

MBean Servers and the MBeanHome Interface

Within a WebLogic Server instance, the actual work of registering and providing access to MBeans is delegated to an MBean Server subsystem. The MBean Server on a Managed Server registers and provides access only to the Local Configuration MBeans and Runtime MBeans on the current Managed Server. The MBean Server on an Administration Server registers and provides access to the domain's Administration MBeans as well as the Local Configuration MBeans and Runtime MBeans on the Administration Server.

Note: On a Managed Server that is configured for Managed Server Independence (MSI), the MBean Server also registers the Administration MBean replicas that the server uses to start if the Administration Server is not available. Do not interact with these Administration MBean replicas. For more information, refer to "Replication of MBeans for Managed Server Independence" on page 1-8.

To access the MBean Server subsystem, you use the `weblogic.management.MBeanHome` interface. From `MBeanHome`, you can use any of the following interfaces to interact with the MBean Server and its MBeans (see Figure 1-10):

- `javax.management.MBeanServer`, which is the standard JMX interface for interacting with MBeans. You can use this interface to look up MBeans that are registered in an MBean Server, determine the set of operations available for an MBean, and determine the type of data that each operation returns. If you invoke MBean operations through the `MBeanServer` interface, you must use standard JMX methods. For example:
  - `MBeanHome.getMBeanServer().getAttribute(`*`MBeanObjectName, attributeName`*`)`
  - `MBeanHome.getMBeanServer().setAttribute(`*`MBeanObjectName, attributeName`*`)`
  - `MBeanHome.getMBeanServer().invoke(`*`MBeanObjectName, operationName, params, signature`*`)`

  For a complete list of `MBeanServer` APIs, refer to view the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

  The `MBeanServer` interface is your only option for interacting with MBeans that you have created and registered (non-WebLogic MBeans).

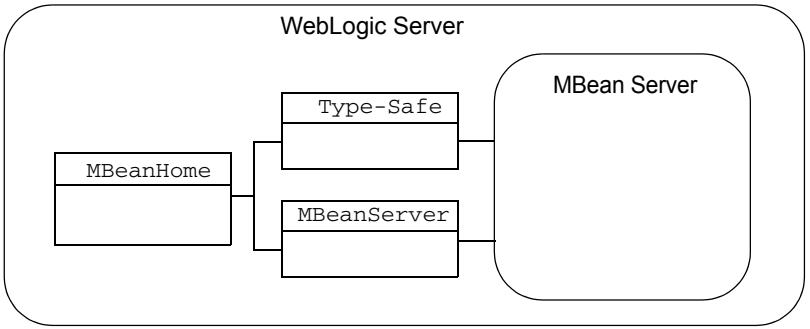
- `weblogic.management.RemoteMBeanServer`, which extends the `javax.management.MBeanServer` and `java.rmi.Remote` interfaces. Use the `RemoteMBeanServer` interface if you want to use standard JMX techniques to access WebLogic Server MBeans from remote JVMs or if you want to interact with non-WebLogic MBeans from a remote JVM.

- A WebLogic Server type-safe interface that makes it appear as though you can invoke an MBean's methods directly. You can use this interface to look up MBeans that are registered in an MBean Server and invoke get, set, and other operations on the MBean. For example:

  ```
  wlMBean = MBeanHome.getMBean(WebLogicObjectName)
  wlMBean.getAttribute
  wlMBean.setAttribute
  wlMBean.operationName
  ```

  The type-safe interface extends the `java.rmi.Remote` interface, so you can use it to access WebLogic Server MBeans from remote JVMs.

Figure 1-10 MBeans Servers and Their Interfaces



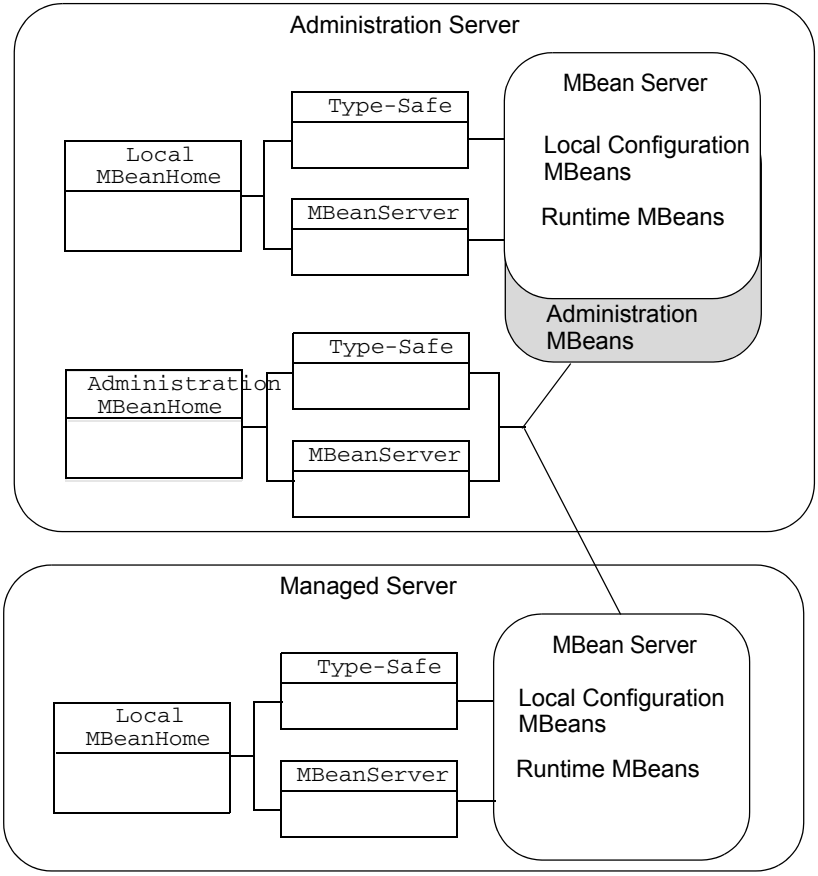
Local MBeanHome and the Administration MBeanHome

All instances of WebLogic Server provide a **local `MBeanHome`** interface through which you can access the MBeans that are hosted in the server instance's MBean Server.

For Managed Servers and Administration Servers, the local `MBeanHome` interface provides access to the Runtime MBeans for the current server only and to all Local Configuration MBeans in the domain.

The Administration Server provides an additional instance of the `MBeanHome` interface. This **Administration `MBeanHome`** provides access to Administration MBeans along with all other MBeans on all server instances in the domain. The Administration `MBeanHome` uses RMI to contact MBeans on Managed Servers, which uses more network resources and might take longer than using a local `MBeanServer` or `MBeanHome` interface. (See Figure 1-11.)

Figure 1-11 Local and Administration MBeanHome Interfaces



The local `MBeanHome` and the Administration `MBeanHome` are two instances of the same interface class, so the APIs for the two types of `MBeanHome` differ only in the name of the `MBeanHome` instance and in the set of MBeans that you can access.

Notifications and Monitoring

Depending on your management needs, you can use MBean APIs to view MBean attributes only upon request, or you can use the WebLogic Server notification and monitoring facilities, which automatically broadcast reports (JMX notifications) when MBean attributes change.

To use these facilities:

- Create a JMX listener, which listens for and reports all attribute changes within an MBean that you specify. For example, you could use a listener with some additional logic to send an email to a System Administrator any time a user changes the configuration of a deployed component. For information about using listeners, refer to Chapter 6, "Using WebLogic Server MBean Notifications and Monitors."
- Create a JMX monitor, which listens for and reports only the changes to specific MBean attributes that fall outside a set of parameters that you set. For example, you could use a monitor with some additional logic to send an email to a System Administrator when the number of open thread pools exceeds a specified limit. For more information, refer to Chapter 6, "Using WebLogic Server MBean Notifications and Monitors."

The Administration Console and the weblogic.Admin Utility

The WebLogic Server Administration Console and the `weblogic.Admin` utility are examples of management utilities that use the WebLogic Server JMX services. You can use these interfaces to familiarize yourself with WebLogic Server management services before developing your JMX applications.

The Administration Console

The Administration Console is a Web application with servlets that invoke the WebLogic Server JMX APIs. Almost all of the values that the Administration Console presents are attributes of Administration MBeans and Runtime MBeans. Because the Administration Console does not read or write Local Configuration MBeans, it is possible that it reports a value that a server instance is not currently using. For example, if you use a `weblogic.Server` startup option to override the configured listen port, the Administration Console reports the value that is in the `config.xml` file, not the overriding value.

To determine which MBean attribute the Administration Console is presenting, click the question mark icon in the top banner. In the help window, click the Attributes link to see the MBean class and attribute that is associated with field on the Administration Console.

The caution icon (yellow triangle with an exclamation point) next to a field in the Administration Console indicates that an attribute is not dynamic. If you modify such an attribute, the underlying managed resource cannot use the new value until you restart the server.

If you modify a dynamic value from the Administration Console, the console updates the corresponding Administration MBean. For information on how this change is propagated to the Local Configuration MBean, refer to "The Life Cycle of Configuration MBeans" on page 1-5.

The weblogic.Admin Utility

The `weblogic.Admin` utility provides several commands that create, get and set values for, invoke operations on, and delete instances of Administration and Configuration MBeans. It also provides commands to get values and invoke operations on Runtime MBeans. You could create shell scripts that use this utility instead of creating JMX applications to programmatically interact with the WebLogic Server management services, however, the performance of a JMX application is superior to a shell script that invokes command-line utilities.

You can also use the `weblogic.Admin` utility to verify object names of MBeans and to get and set attributes from a command line before committing to writing JMX code. Subsequent sections in this document provide examples of using the `weblogic.Admin` utility as part of your JMX development.

For more information, refer to "Commands for Managing WebLogic Server MBeans" in the *WebLogic Server Command Line Reference*.

CHAPTER 2

Accessing WebLogic Server MBeans

All JMX tasks—viewing or changing MBean attributes, using notifications, and monitoring changes—use the same process to access MBeans.

The following sections describe how to access WebLogic Server MBeans:

Accessing MBeans: Main Steps

The main steps for accessing MBeans in WebLogic Server are as follows:

1. Use a `weblogic.management.MBeanHome` interface to access the MBean Server. See "Accessing an MBeanHome Interface" on page 2-4.
2. Use one of the following interfaces to retrieve, look up, and invoke operations on MBeans:
   - A type-safe interface that WebLogic Server provides. This interface, which is a WebLogic Server extension to JMX, can retrieve and invoke operations only on the MBeans that WebLogic Server provides. See "Using the Type-Safe Interface to Access MBeans" on page 2-10.
   - The standard JMX `javax.management.MBeanServer` interface, which can retrieve and invoke operations on WebLogic Server MBeans or on MBeans that you create. See "Using the MBeanServer Interface to Access MBeans" on page 2-18.
   - The `weblogic.management.RemoteMBeanServer` interface, which extends the `javax.management.MBeanServer` and `java.rmi.Remote` interfaces.

   In most cases, you use these interfaces to retrieve a list of MBeans and then filter the list to retrieve and invoke operations on a specific MBean. However, if you know the `WebLogicObjectName` of an MBean, you can retrieve an MBean directly by name.

Determining Which Interfaces to Use

When accessing MBeans, you must make two choices about which interfaces you use:

- Whether to use the `MBeanHome` interface on a local server instance or the Administration `MBeanHome` interface to access the MBean Server. The `MBeanHome` interface that you choose determines the set of MBeans you can access.

  The following table lists typical considerations for determining whether to use the local `MBeanHome` interface or the Administration `MBeanHome` interface.

Table 2-1 Deciding Between the Local or Administration MBeanHome

If your application manages...	Retrieve this `MBeanHome` interface...
Local Configuration MBeans or Runtime MBeans	Administration `MBeanHome` or local `MBeanHome` The Administration `MBeanHome` provides a single, convenient interface from which to access all MBeans on all server instances in a domain. When you use this interface, you typically retrieve MBeans from multiple server instances and then iterate through the list to find an MBean for a specific server instance. A local `MBeanHome` provides access to the Runtime MBeans for the current server only and to all Local Configuration MBeans in the domain. The interface uses fewer network hops to access MBeans because it requires your client to establish a direct connection to the server instance. When using a local `MBeanHome`, you typically retrieve one of several top-level MBeans and use them to walk the MBean hierarchy. See "Walking the Hierarchy of Local Configuration and Runtime MBeans" on page 2-14.
Administration MBeans	Administration `MBeanHome`

- Whether to use the WebLogic Server type-safe interface, the standard JMX `MBeanServer` interface, or the WebLogic `RemoteMBeanServer` interface to access and invoke operations on MBeans.

  The following table lists typical considerations for determining whether to use the type-safe interface or the `MBeanServer` interface.

Table 2-2 Deciding Between the Type-Safe Interface or the MBeanServer Interface

If your application...	Use this interface...
Interacts only with WebLogic Server MBeans.	The WebLogic Server type-safe interface
Might need to run on J2EE platforms other than WebLogic Server	`MBeanServer` If your client accesses MBeans that are running in a separate JVM, use `RemoteMBeanServer`. Your client code will still be portable to other J2EE servers, although you cannot on other J2EE servers you must substitute `RemoteMBeanServer` with some other interface that extends the standard `MBeanServer` interface.
Interacts with non-WebLogic Server MBeans	`MBeanServer` If your client accesses MBeans that are running in a separate JVM, use `RemoteMBeanServer`.

Accessing an MBeanHome Interface

The simplest process for retrieving a local `MBeanHome` interface or an Administration `MBeanHome` interface is to use the WebLogic Server `Helper` class. If you are more comfortable with a standard J2EE approach, you can use the Java Naming and Directory Interface (JNDI) to retrieve `MBeanHome`.

Using the Helper APIs to Retrieve an MBeanHome Interface

WebLogic Server provides the `weblogic.management.Helper` APIs to simplify the process of retrieving `MBeanHome` interfaces.

To use the `Helper` APIs, collect the following information:

- The username and password of a WebLogic Server user who has permission to invoke MBean operations. For more information, refer to "Security Roles" in the *Securing WebLogic Resources* guide.
- If you are accessing a local `MBeanHome` interface, the name of the target server (as defined in the domain configuration) and the URL of the target server.
- If you are accessing the Administration `MBeanHome`, the URL of the Administration Server.

After you collect the information, use one of the following APIs:

- To retrieve a local `MBeanHome`:
  `Helper.getMBeanHome(java.lang.String user, java.lang.String password, java.lang.String serverURL, java.lang.String serverName)`
- To retrieve the Administration `MBeanHome`:
  `Helper.getAdminMBeanHome(java.lang.String user, java.lang.String password, java.lang.String adminServerURL)`

For more information about the `Helper` APIs, refer to the WebLogic Server Javadoc.

Example: Retrieving a Local MBeanHome Interface

The following example (Listing 2-1) is a class that uses the `Helper` API to obtain the local `MBeanHome` interface for a server named `MS1`.

Listing 2-1 Retrieving a Local MBeanHome Interface

```
import weblogic.management.Helper;
import weblogic.management.MBeanHome;

public class UseHelper {
    public static void main(String[] args) {
        String url = "t3://localhost:7001";
        String username = "weblogic";
        String password = "weblogic";
        String msName = "MS1";
        MBeanHome localHome = null;

        try {
            localHome = (MBeanHome)Helper.getMBeanHome(username, password, url,
                        msName);
            System.out.println("Local MBeanHome for" + localHome +
                        " found using the Helper class");
        } catch (IllegalArgumentException iae) {
            System.out.println("Illegal Argument Exception: " + iae);
        }
    }
}
```

Using JNDI to Retrieve an MBeanHome Interface

While the `Helper` APIs provide a simple way to obtain an `MBeanHome` interface, you might be more familiar with the standard approach of using JNDI to retrieve the `MBeanHome`. From the JNDI tree of a Managed Server, you can access the server's local `MBeanHome` interface. From the JNDI tree of the Administration Server, you can access the Administration `MBeanHome` as well as the local `MBeanHome` interface for any server instance in the domain.

To use JNDI to retrieve an `MBeanHome` interface:

1. Construct a `weblogic.jndi.Environment` object and use `Environment` methods to configure the object:

   a. Use the `setSecurityPrincipal` and `setSecurityCredentials` methods to specify user credentials.

      WebLogic Server verifies that the user credentials you supply have been granted permission to carry out requests through the `MBeanHome` interface. For more information, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

   b. If your application and the `MBeanHome` interface are in different JVMs, use the `Environment.setProviderUrl` method to specify the server instance that hosts the `MBeanHome` interface. The URL must specify the listen address of the server and the port on which the server listens for administrative requests.

      If you want to retrieve the Administration MBeanHome, `setProviderUrl` must specify the Administration Server.

   c. Use the `getInitialContext` method to initialize a `javax.naming.Context` object.

   For example, the following lines of code set the initial context to a server instance that runs on a host computer named `WLServerHost` and uses the default domain-wide administration port to receive administrative requests:

```
Environment env = new Environment();
    env.setProviderUrl("t3://WLServerHost:9002");
    env.setSecurityPrincipal("weblogic");
    env.setSecurityCredentials("weblogic");
    Context ctx = env.getInitialContext();
```

   For more information about `weblogic.jndi.Environment`, refer to the WebLogic Server Javadoc.

2. Use `javax.naming.Context` methods to look up and retrieve the `MBeanHome` interface for the current context.

   Use one of the following APIs, depending on whether you are retrieving a local `MBeanHome` interface or the Administration MBeanHome:

   – To retrieve the local `MBeanHome` for the current context, use the following API:
   `javax.naming.Context.lookup(MBeanHome.LOCAL_JNDI_NAME)`

   – If the current context is an Administration Server, use the following API to retrieve the local `MBeanHome` of any server instance in the domain:
   `javax.naming.Context.lookup("weblogic.management.home.`*`relevantServerName`*`")`

   where *`relevantServerName`* is the name of a server as defined in the domain configuration.

   – If the current context is an Administration Server, use the following API to retrieve the Administration `MBeanHome`:
   `javax.naming.Context.lookup(MBeanHome.ADMIN_JNDI_NAME)`

   The Administration `MBeanHome` interface provides access to all Local Configuration, Administration, and Runtime MBeans in the domain.

   For more information about `javax.naming.Context.lookup(String name)`, refer to the Sun Javadoc.

The following sections are examples of retrieving `MBeanHome` interfaces:

- Example: Retrieving the Administration MBeanHome from an External Client
- Example: Retrieving a Local MBeanHome from an Internal Client

Example: Retrieving the Administration MBeanHome from an External Client

The following example (Listing 2-2) shows how an application running in a separate JVM looks up the Administration `MBeanHome` interface. In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 2-2 Retrieving the Administration MBeanHome from an External Client

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.AuthenticationException;
import javax.naming.CommunicationException;
import javax.naming.NamingException;
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;

public class RetrieveMBeanHome{

    public static void main(String[] args) {
        MBeanHome home = null;
        //domain variables
        String url = "t3://localhost:7001";
        String username = "weblogic";
        String password = "weblogic";

        //Setting an initial context.
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //Retrieving the Administration MBeanHome interface
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Admin MBeanHome: " + home + " from the
                               Admin server");

        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }
    }
}
```

Example: Retrieving a Local MBeanHome from an Internal Client

If your client application resides in the same JVM as the Administration Server (or the WebLogic Server instance you want to manage), the JNDI lookup for the `MBeanHome` is simpler. Listing 2-3 shows how a servlet running in the same JVM as the Administration Server would look up the local `MBeanHome` for a server instance named `myserver`.

Listing 2-3 Retrieving a Local MBeanHome from an Internal Client

```
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import javax.naming.Context;

public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException{
        doPost(req,res);
    }

    public void doPost(HttpServletRequest req,HttpServletResponse res)
    throws ServletException{

        try {
            Environment env = new Environment();
            env.setProviderUrl("t3://localhost:7001");
        env.setSecurityPrincipal("weblogic");
            env.setSecurityCredentials("weblogic");

            //Setting the initial context
            Context ctx = env.getInitialContext();

             //Retrieving the server-specific MBeanHome interface
            MBeanHome home = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Server-specific MBeanHome: " + home);

        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }
    }
}
```

Using the Type-Safe Interface to Access MBeans

After you retrieve the `MBeanHome` interface, the easiest approach for accessing MBeans is to use methods in the `MBeanHome` interface that retrieve a type-safe interface for MBeans.

You can use this type-safe interface only with the MBeans that WebLogic Server provides. You cannot use this type-safe interface for MBeans that are based on MBean types that you create.

Retrieving a List of All MBeans

You can use the `MBeanHome.getAllMBeans` method to look up the object names of MBeans that are within the scope of the `MBeanHome` interface that you retrieve. For example, if you retrieve the Administration `MBeanHome`, using `getAllMBeans()` returns a list of all MBeans in the domain. If you retrieve a Local `MBeanHome` interface, using `getAllMBeans()` returns a list of the Runtime MBeans for the current server only and of all Local Configuration MBeans in the domain.

The example class in Listing 2-4:

1. Uses `JNDI` APIs to retrieve the Administration `MBeanHome` interface.
2. Uses the `MBeanHome.getAllMBeans` method to retrieve all MBeans in a domain.
3. Assigns the list of MBeans to a `Set` object and uses methods of the `Set` and `Iterator` interfaces to iterate through the list.
4. Uses the `WebLogicMBean.getObjectName` method to retrieve the `WebLogicObjectName` of each MBean.
5. Uses the `WebLogicObjectName.getName` and `getType` methods to retrieve the `Name` and `Type` values of the `WebLogicObjectName`

In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 2-4 Retrieving All MBeans in a Domain

```
import javax.naming.Context;
import java.util.Set;
import java.util.Iterator;
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicMBean;
import weblogic.management.WebLogicObjectName;

public class ListAllMBeans{
    public static void main(String args[]) {
        String url = "t3://localhost:7001";
        String username = "weblogic";
        String password = "weblogic";

        try {
            //Obtaining an MBeanHome Using JNDI
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();
            MBeanHome home = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);

            Set allMBeans = home.getAllMBeans();
            System.out.println("Size: " + allMBeans.size());
            for (Iterator itr = allMBeans.iterator(); itr.hasNext(); ) {
                WebLogicMBean mbean = (WebLogicMBean)itr.next();
                WebLogicObjectName objectName = mbean.getObjectName();
                System.out.println(objectName.getName() + " is a(n) " +
                                   mbean.getType());
            }
        }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

For more information about the `MBeanHome.getAllMBeans` method, refer to the WebLogic Server Javadoc.

Retrieving MBeans By Type and Selecting From the List

Instead of retrieving a list of all MBeans in the scope of `MBeanHome`, you can retrieve a list of MBeans that match a specific type. `Type` indicates the type of resource that the MBean manages and whether the MBean is an Administration, Local Configuration, or Runtime MBean. For more information about types of MBeans, refer to the next section, "WebLogic Server Management Namespace" on page 3-1.

The example class in Listing 2-5:

1. Uses JNDI to retrieve the Administration `MBeanHome` interface.
2. Uses the `MBeanHome.getMBeansByType` method to retrieve a list of all `ServerRuntime` MBeans in a domain.
3. Assigns the list of MBeans to a `Set` object and uses methods of the `Set` and `Iterator` interfaces to iterate through the list.
4. Uses the `ServerRuntime.getName` method to retrieve the name of each `ServerRuntime` MBean. The name of a `ServerRuntime` MBean corresponds to the name of a server instance.
5. When it finds the `ServerRuntime` MBean for a server named `Server1`, it prints a message to standard out.

In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 2-5 Selecting by Type from a List of MBeans

```
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.management.ObjectName;

import weblogic.management.MBeanHome;
import weblogic.management.WebLogicMBean;
import weblogic.management.WebLogicObjectName;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.runtime.ServerRuntimeMBean;
import weblogic.jndi.Environment;

public class serverRuntimeInfo {

    public static void main(String[] args) {

        MBeanHome home = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "Server1";
        String username = "weblogic";
        String password = "weblogic";

        ServerRuntimeMBean serverRuntime = null;
        Set mbeanSet = null;
        Iterator mbeanIterator = null;

        //Using JNDI to retrieve the Administration MBeanHome
        //Setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //Getting the Administration MBeanHome
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Admin MBeanHome: " + home );
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        //Using the getMBeansByType method to get all ServerRuntime MBeans
        //in the domain.
```

```
        try {
            mbeanSet = home.getMBeansByType("ServerRuntime");

            //Iterating through the results and comparing the server names
            //find the one we want.
            mbeanIterator = mbeanSet.iterator();
            while(mbeanIterator.hasNext()) {
                serverRuntime = (ServerRuntimeMBean)mbeanIterator.next();
                //Using serverRuntime.getName to find the ServerRuntime
                //MBean for Server1.
                if(serverRuntime.getName().equals(serverName)) {
                    System.out.println("Got the serverRuntimembean: " +
                    serverRuntime + " for: " + serverName);
                 }
            }
         } catch (Exception e) {
               System.out.println("Exception caught: " + e);
         }
    }
}
```

For more information about the `MBeanHome.getMBeansByType` method, refer to the WebLogic Server Javadoc.

Walking the Hierarchy of Local Configuration and Runtime MBeans

WebLogic Server MBeans exist within a hierarchy that reflects the resources with which they are associated. For example, each server instance can contain multiple execute queues, and WebLogic Server represents this relationship by making each `ExecuteQueueMBean` a child of a `ServerMBean`.

Walking the hierarchy of MBeans is the easiest way to retrieve Local Configuration and Runtime MBeans. If you want to retrieve Administration MBeans, or if you want to use the Administration `MBeanHome` to retrieve MBeans, BEA recommends that you retrieve MBeans by type and then filter the list. See "Retrieving MBeans By Type and Selecting From the List" on page 2-12.

The root of the configuration MBean hierarchy is `DomainMBean`. Below this root are MBeans such as:

- `ClusterMBean`
- `ServerMBean`

- `ApplicationMBean`
- `RealmMBean`
- JDBC and JMS configuration MBeans

The root of the runtime hierarchy is `ServerRuntimeMBean`. Just below this root are MBeans such as:

- `ClusterRuntimeMBean`
- `ApplicationRuntimeMBean`
- JDBC and JMS runtime MBeans

Parent MBeans usually provide methods for retrieving their children. For example, `ServerMBean.getExecuteQueues` returns all `ExecuteQueueMBeans` that have been configured for the server.

For more information about the hierarchy, see Chapter 3, "WebLogic Server Management Namespace."

To walk the hierarchy of Local Configuration MBeans or Runtime MBeans:

1. From your JMX application, retrieve the local `MBeanHome` interface.
2. From the local `MBeanHome` interface, retrieve one of the top-level MBeans by invoking one of the following methods:
   - `getConfigurationMBean (java.lang.String name, java.lang.String type)`

     See the Javadoc for MBeanHome.getConfigurationMBean.
   - `getRuntimeMBean (java.lang.String name, java.lang.String type)`

     See the Javadoc for MBeanHome.getRuntimeMBean.

   Use these methods to retrieve only MBeans that are immediately below `DomainMBean` or `ServerRuntimeMBean`. These methods do not return MBeans that are below the first level of the MBean hierarchy.
3. From the MBean that you retrieved, invoke methods to retrieve the MBean's children.

   If a parent MBean does not provide methods to retrieve child MBeans, use `getMBeanByType()` and iterate over the results to find the MBean that matches your criteria. If you want to retrieve Local Configuration MBeans, be sure to append `Config` to the MBean type value. See "Retrieving MBeans By Type and Selecting From the List" on page 2-12.

Note: BEA recommends that you retrieve Local Configuration MBeans only to read values; do not **change** attribute values in Local Configuration MBeans. When the Managed Server replicates the data of other Managed Servers, it uses the values that are stored in Administration MBeans. Communication problems can occur if the values in Administration MBeans and Local Configuration MBeans differ.

Listing 2-6 is an example of retrieving all Local Configuration `ExecuteQueueMBeans` on a server instance named `MedRecServer`.

Listing 2-6 Retrieving Local Configuration ExecuteQueueMBeans

```
import javax.naming.Context;
import javax.management.ObjectName;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicMBean;
import weblogic.management.WebLogicObjectName;
import weblogic.management.configuration.ConfigurationMBean;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.configuration.ExecuteQueueMBean;

import weblogic.jndi.Environment;

public class serverConfigInfo {
    public static void main(String[] args) {
        MBeanHome home = null;
        ServerMBean servercfg = null;
        ExecuteQueueMBean[] xqueues = null;
        ExecuteQueueMBean xqueue = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "MedRecServer";
        String username = "weblogic";
        String password = "weblogic";

        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);

            //Setting the initial context
            Context ctx = env.getInitialContext();

            //Retrieving the server-specific MBeanHome interface
            home = (MBeanHome)ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);
            System.out.println("Got the Server-specific MBeanHome: " + home);
```

```
            //Retrieving the Local Configuration ServerMBean
             servercfg = (ServerMBean)home.getConfigurationMBean(serverName,
                                                    "ServerConfig");
            System.out.println("Got the Server Config MBean: " + servercfg);

            //Retrieving all ExecuteQueue MBeans that have been
            //configured for the server instance
            xqueues = servercfg.getExecuteQueues();

            //Iterating through the results
            for (int i=0; i < xqueues.length; i++){
                 xqueue = xqueues[i];
                 System.out.println("Execute queue name: " +
                     xqueue.DEFAULT_QUEUE_NAME);
                System.out.println("Thread count:" + xqueue.getThreadCount());

              }
        } catch (Exception e) {
             System.out.println("Exception caught: " + e);
         }
    }
}
```

If you want to create generic JMX code that you can run on any server instance to retrieve its Server Configuration MBean:

1. From the local `MBeanHome` interface, use the `getMBeansByType` method to retrieve the server's `ServerRuntimeMBean`:
   `serverRuntime = MBeanHome.getMBeansByType(ServerRuntime)`

   The local `MBeanHome` interface can access only the runtime MBeans that are specific to the current server instance, so `getMBeansByType(ServerRuntime)` returns only the `ServerRuntimeMBean` for the current server.

2. Use `ServerRuntimeMBean`'s `getName` method to retrieve the name of the server:
   `serverName = serverRuntime.getName()`

3. Use the server name when invoking `MBeanHome.getConfigurationMBean`:
   `MBeanHome.getConfigurationMBean(serverName,"ServerConfig")`

For more information, see "Example: Determining the Active Domain and Servers" on page 5-1.

Using the MBeanServer Interface to Access MBeans

A standard JMX approach for interacting with MBeans is to use the `javax.management.MBeanServer` interface to look up the MBeans that are registered in the MBean Server. Then you use the `MBeanServer` interface to get or set MBean attributes or to invoke MBean operations. For the complete list of `MBeanServer` methods, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

You can use the following techniques to retrieve the `MBeanServer` interface:

- Use the `getMBeanServer()` method in the `weblogic.management.MBeanHome` interface.

  Use this technique if your JMX client already has a reference to the `MBeanHome` interface. See the Javadoc for MBeanHome.getMBeanServer().

- Look up the `javax.management.MBeanServer` interface from the WebLogic Server JNDI tree.

  Use this technique if you do not want to import WebLogic Server classes into your JMX client. Each server instance publishes it `MBeanServer` interface under the following JNDI name: `weblogic.management.server`.

The example code in Listing 2-7 looks up the `MBeanServer` interface from a server's JNDI tree. To establish an initial context in a WebLogic Server JNDI tree, a client must specify the server's connection information, the name of the WebLogic Server context factory, and the WebLogic Server login credentials. See the Javadoc for javax.naming.Context.

In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 2-7 Retrieving MBeanServer Through JNDI

```
String url = "t3://localhost:7001"; //URL of the server instance
String username = "weblogic";
String password = "weblogic";
MBeanServer rmbs = null;

Hashtable props = new Hashtable();
props.put(Context.PROVIDER_URL, url);
props.put(Context.INITIAL_CONTEXT_FACTORY,
   "weblogic.jndi.WLInitialContextFactory");
   props.put(Context.SECURITY_PRINCIPAL, username);
   props.put(Context.SECURITY_CREDENTIALS, password);

InitialContext ctx = new InitialContext(props);
rmbs = (MBeanServer) ctx.lookup("weblogic.management.server");
```

CHAPTER 3

WebLogic Server Management Namespace

When you instantiate a WebLogic Server MBean, the MBean Server subsystem registers the instance under a name that conforms to the `weblogic.management.WebLogicObjectName` conventions. These naming conventions create a hierarchical JMX namespace that you use when looking up MBeans. (For more information about the MBean Server subsystem, see "MBean Servers and the MBeanHome Interface" on page 1-13.)

The following sections describe the WebLogic Server management namespace:

Conventions for WebLogicObjectName

`WebLogicObjectName` is a subclass of `javax.management.ObjectName`. To provide unique names for MBeans, all JMX object names consist of two parts:

- A JMX domain name, which is a case-sensitive string that defines a top level within the JMX namespace.

  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.

- An unordered set of one or more name-value pairs (key properties).

  The key properties create unique object names within a given JMX domain. They do not need to reflect attributes within the MBean that they name.

The MBean's `WebLogicObjectName` uses the following conventions to provide a unique identification for a given MBean:

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

The order of the *attribute*`=`*value* pairs is not significant, but the name must begin with *domain*`:`.

The following table describes each name component.

Table 3-1 WebLogic Server MBean Naming Conventions

This Component	Specifies
domain:	The name of the JMX domain. By convention, the JMX domain name for all WebLogic Server MBeans corresponds to the name of the WebLogic Server administration domain.
Name=*name*	The string that you provided when you created the resource that the MBean represents. For example, when you create a JDBC connection pool, you must provide a name for that pool, such as `MyPool1`. The `JDBCConnectionPoolMBean` that represents `MyPool1` uses `Name=MyPool1` in its JMX object name. The `WebLogicObjectName.getName` method returns this value for any given MBean. 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 interface class of which the MBean is an instance. All WebLogic Server MBeans are an instance of one of the interface classes defined in the `weblogic.management.configuration` or `weblogic.management.runtime` packages. For Configuration MBeans, `Type` also refers to whether an instance is an Administration MBean or a Local Configuration MBean. For a complete list of all WebLogic Server MBean interface classes, refer to the WebLogic Server Javadoc for the `weblogic.management.configuration` or `weblogic.management.runtime` packages. To determine the value that you provide for the `Type` component: 1. Find the MBean's interface class and remove the `MBean` suffix from the class name. For example, for an MBean that is an instance of the `weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean`, use `JDBCConnectionPoolRuntime`. 2. For a Local Configuration MBean, append `Config` to the name. For example, for a Local Configuration MBean that is an instance of the `weblogic.management.configuration.JDBCConnectionPoolMBean` interface class, use `JDBCConnectionPoolConfig`. For the corresponding Administration MBean instance, use `JDBCConnectionPool`.

Table 3-1 WebLogic Server MBean Naming Conventions

This Component	Specifies
`Location=`*`servername`*	All Runtime and Local Configuration MBeans include a `Location` component that specifies the name of the server on which that MBean is located. Administration MBeans do not include this component. For example, for a server instance named myserver, there are two instances of `ServerMBean`: • The Administration MBean, whose object name is `mydomain:Name=myserver,Type=Server` • The Local Configuration MBean, whose object name is: `mydomain:Name=myserver,Type=Server,Location=myserver` For information about accessing these MBean instances, see "Determining Which Interfaces to Use" on page 2-3. The `WebLogicObjectName.getLocation` method returns this value for any given MBean.

Table 3-1 WebLogic Server MBean Naming Conventions

This Component	Specifies

TypeOfParentMBean=*NameOfParentMBean*	To create a hierarchical namespace, Runtime, Local Configuration, or Administration 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.
	
	By convention, WebLogic Server child MBeans 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 `WebLogicObjectName`:
	
	`medrec:Name=MedRecServer,Type=Log`
	
	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 `WebLogicObjectName` for an `EJBComponentRuntime` MBean for in the MedRec sample application:
	`medrec:ApplicationRuntime=MedRecServer_MedRecEAR, Location=MedRecServer,Name=MedRecServer_MedRecEAR_Session EJB,ServerRuntime=MedRecServer,Type=EJBComponentRuntime`
	
	The `ApplicationRuntime=MedRecServer_MedRecEAR` attribute/value pair indicates that the EJB instance is a module within the MedRec enterprise application and a child of the `MedRecServer_MedRecEAR` ApplicationRuntimeMBean. The `ServerRuntime=MedRecServer` attribute/value pair indicates that the EJB instance is currently deployed on a server named MedRecServer and a child of the `MedRecServer` ServerRuntimeMBean.
	
	See:
	• "Locating Administration MBeans Within the Namespace" on page 3-6
	• "Using weblogic.Admin to Find the WebLogicObjectName" on page 3-21

Conventions for Security-Provider MBean Names

While the MBeans that you use to manage security providers use JMX object names, they do not use names of type `weblogic.management.WebLogicObjectName`. Instead, the security providers that are included with WebLogic Server use the following JMX-compliant naming conventions:

`Security:Name=`*`realmNameProviderName`*

In this convention, `Security:` is the name of the JMX domain, *`realmName`* is the name of a security realm and *`ProviderName`* is the name that you give to the security provider. For example, the name of the MBean for the authentication provider that WebLogic Server installs is `Security:Name=myrealmDefaultAuthenticator`.

BEA recommends that you follow this convention for any additional security providers that you configure. If you use the Administration Console to add a security provider to the realm, your security-provider MBean names will follow the recommended naming convention.

For more information about security providers, see Developing Security Providers for WebLogic Server.

Locating Administration MBeans Within the Namespace

System administrators frequently use JMX APIs, the `weblogic.Admin` utility, or the `wlconfig` Ant task to automate the creation of resources within a WebLogic Server domain. To successfully configure these resources, you must create Administration MBeans and locate them within the namespace hierarchy.

Note: The management namespace for Local Configuration MBeans and Runtime MBeans is also hierarchical; however, because system administrators infrequently use APIs or other command-line utilities to access these types of MBeans, their namespace is not documented.

The following sections describe the namespace for the Administration MBeans that configure many WebLogic Server resources and server attributes:

- "Server Communication and Protocols Configuration Namespace" on page 3-7
- "Domain and Server Logging Configuration Namespace" on page 3-9
- "Applications Configuration Namespace" on page 3-10
- "Security Configuration Namespace" on page 3-12
- "JDBC Configuration Namespace" on page 3-15
- "JMS Configuration Namespace" on page 3-16
- "Clusters Configuration Namespace" on page 3-19
- "Machines and Node Manager Configuration Namespace" on page 3-20

Note: With the exception of `DomainMBean`, all MBeans are direct or indirect children of the domain's `DomainMBean`. Because this parent-child relationship applies to all MBeans, it is not expressed in MBean object names.

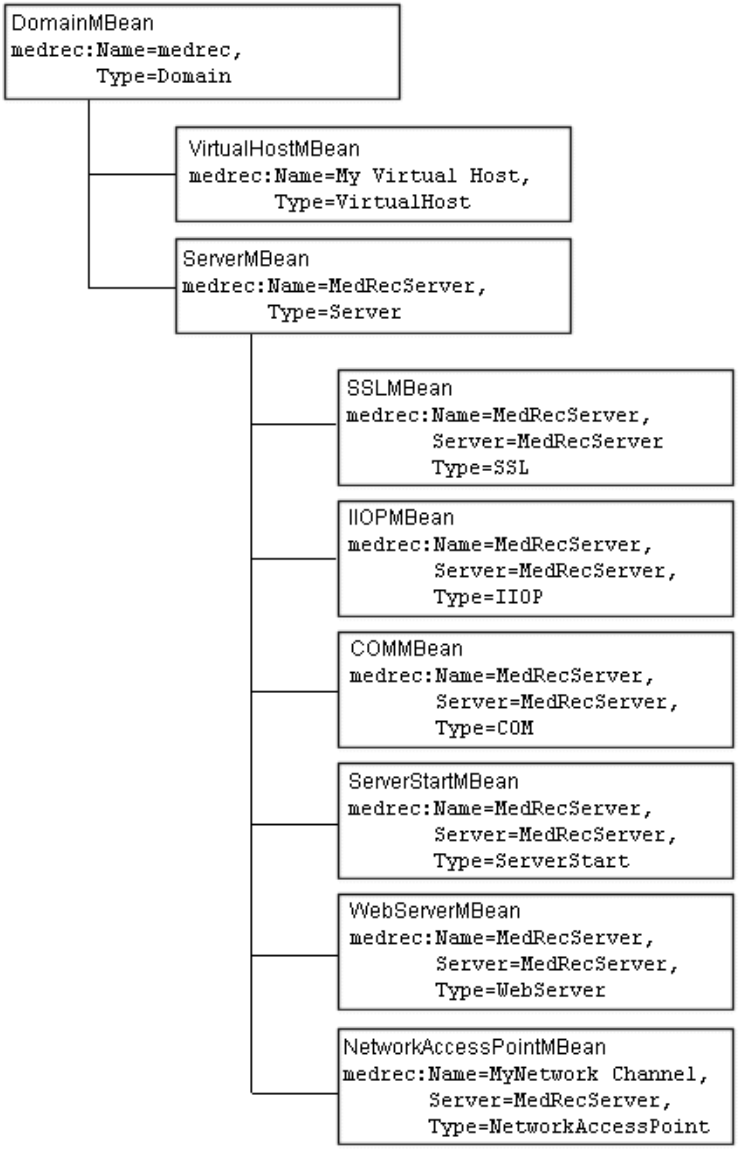
Server Communication and Protocols Configuration Namespace

A WebLogic Server instance uses attributes from several MBeans to determine how it communicates with clients and other servers. Table 3-2 introduces the MBeans and Figure 3-1 illustrates the namespace in the sample MedRec domain.

Table 3-2 MBeans for Server Communication and Protocols

This MBean...	Configures...
`ServerMBean`	Listen address, listen port, and enables protocols and tunneling. See `ServerMBean` Javadoc.
`SSLMBean`	The SSL protocol. See `SSLMBean` Javadoc.
`IIOPMBean`	The IIOP protocol. See `IIOPMBean` Javadoc.
`COMMBean`	The COM protocol. See `COMMBean` Javadoc.
`WebServerMBean`	The HTTP and HTTPS protocols. See `WebServerMBean` Javadoc.
`ServerStartMBean`	Arguments that a Node Manager uses to start this server instance. See `ServerStartMBean` Javadoc.
`NetworkAccessPointMBean`	Additional network connections (network channel). See `NetworkAccessPointMBean` Javadoc.
`VirtualHostMBean`	Host names to which the server responds. See `VirtualHostMBean` Javadoc.

Figure 3-1 MBean Namespace for Server Communication and Protocols



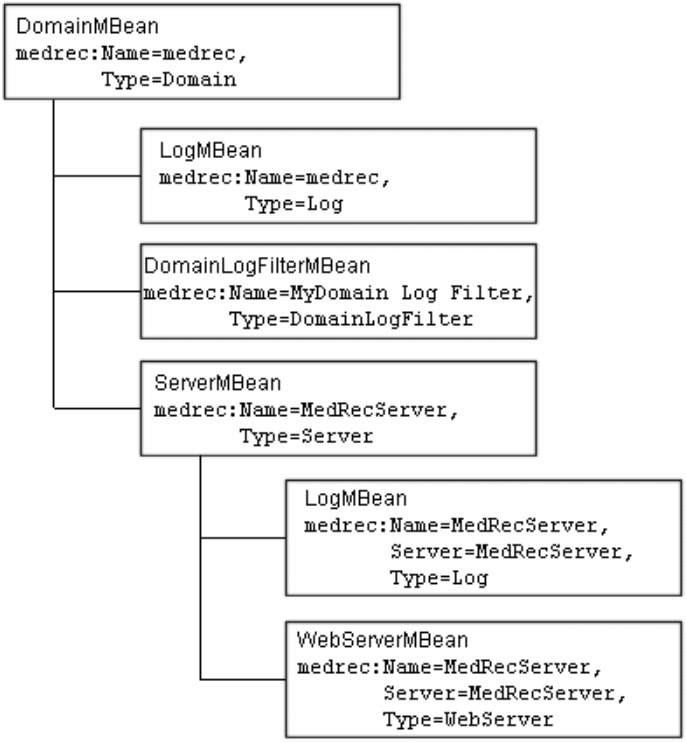
Domain and Server Logging Configuration Namespace

Within a WebLogic Server domain, several MBeans configure logging services. Table 3-3 introduces the MBeans and Figure 3-2 illustrates the namespace in the sample MedRec domain.

Table 3-3 MBeans for Domain and Server Logging

This MBean...	Configures...
`LogMBean`	Log file names and rotation criteria. The Administration Server maintains an instance of `LogMBean` for the domain-wide message log, and each server instance maintains its own instance for its local server log. See `LogMBean` Javadoc.
`DomainLogFilterMBean`	A domain log filter, which determines which messages a server instance sends to the domain-wide message log. Each domain log filter is represented by its own instance of `DomainLogFilterMBean`. See `DomainLogFilterMBean` Javadoc.
`ServerMBean`	JDBC and JTA logging. Also determines which domain log filter the server instance uses. See `ServerMBean` Javadoc.
`WebServerMBean`	Logging HTTP requests. See `WebServerMBean` Javadoc.

Figure 3-2 Logging MBeans



Applications Configuration Namespace

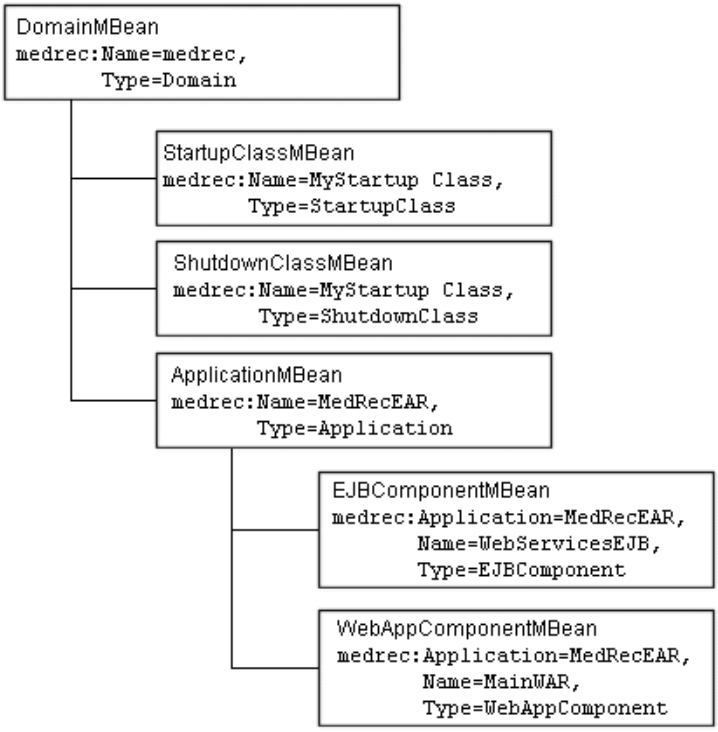
When you target and deploy J2EE modules (enterprise applications, Web applications, or EJBs), WebLogic Server creates MBeans for managing the module's configuration.

In addition, WebLogic Server creates MBeans for the startup classes and shutdowns classes that you configure and target. Table 3-4 introduces the MBeans and Figure 3-3 illustrates the namespace in the sample MedRec domain.

Table 3-4 MBeans for Applications

This MBean...	Configures...
`StartupClassMBean`	A startup class and the server instances to which the class is targeted. See `StartupClassMBean` Javadoc.
`ShutdownClassMBean`	A shutdown class and the server instances to which the class is targeted. See `ShutdownClassMBean` Javadoc.
`ApplicationMBean`	Deployment options for the Web applications and EJBs that it contains. Each application is managed by its own instance of `ApplicationMBean`. See `ApplicationMBean` Javadoc.
`WebAppComponentMBean`	A Web application. See `WebAppComponentMBean` Javadoc.
`EJBComponentMBean`	Location and deployment information for all EJBs in an EJB JAR file. See `EJBComponentMBean` Javadoc.

Figure 3-3 Application MBeans



Security Configuration Namespace

Within a WebLogic Server domain, each security realm is managed by its own instance of `RealmMBean` and several MBeans for the security providers that are configured for the realm.

Because each security realm can be customized, the `weblogic.management.security.RealmMBean` and security-provider MBeans occupy a separate management namespace from all other WebLogic Server MBeans. In addition, while the `RealmMBean` and security-provider MBean names are valid JMX object names, they do not follow `WebLogicObjectName` conventions. Instead, BEA recommends the following naming convention for any security realm and security-provider MBeans you create:

`Security:`*`realmnameProviderName`*

If you use the Administration Console to configure your realm and security providers, the MBean names follow the recommended convention.

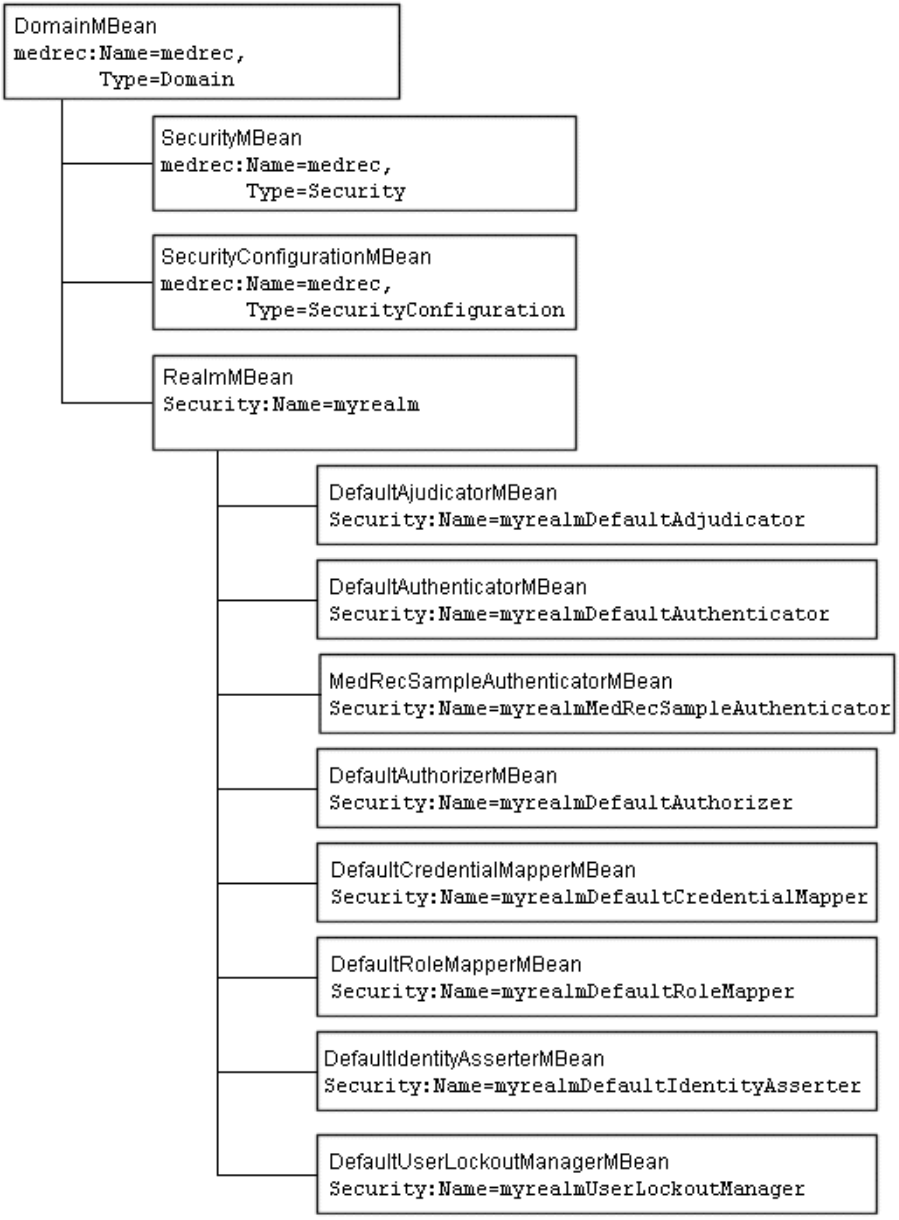
In addition to realm and security-provider MBeans, a WebLogic Server domain uses the MBeans in Table 3-5 to configure security. These MBeans are in the same namespace as other WebLogic Server MBeans.

Table 3-5 MBeans for Security

This MBean...	Configures...
`SecurityMBean`	Domain-wide security configuration information. See `SecurityMBean` Javadoc.
`SecurityConfigurationMBean`	The security realm, password policy, and connection filter that the domain uses. See `SecurityConfigurationMBean` Javadoc.
`weblogic.management.configuration.RealmMBean`	This `RealmMBean`, which is in a different package from the `weblogic.management.security.RealmMBean` described above, is deprecated and used to configure realms that use compatibility security.

Figure 3-4 illustrates the security namespace in the sample MedRec domain. The security realm and providers in Figure 3-4 are those that WebLogic Server installs by default. In a domain that you create, your security namespace might look different depending on the realm and security providers that you configure.

Figure 3-4 Security MBeans



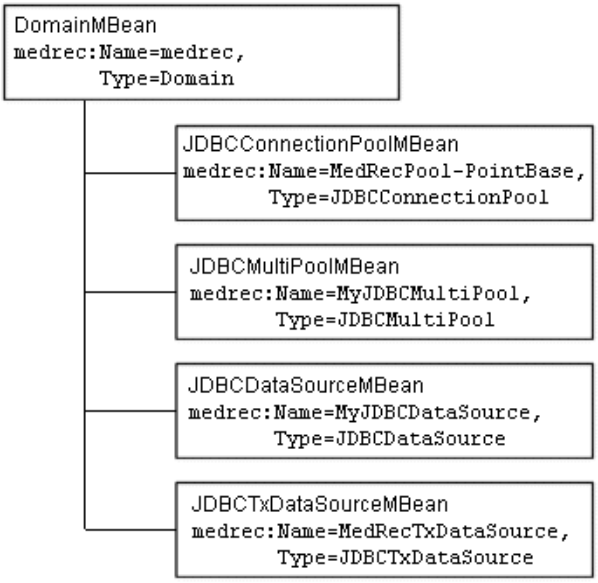
JDBC Configuration Namespace

WebLogic Server uses several MBeans to provide a management interface for JDBC services. The names of JDBC MBeans do not reflect a hierarchy because all JDBC MBeans are direct children of the `DomainMBean`. Table 3-6 introduces the MBeans and Figure 3-5 illustrates the namespace in the sample MedRec domain. Figure 3-5 adds MBeans for the JDBC features that the MedRec domain does not use.

Table 3-6 MBeans for JDBC

This MBean...	Configures...
`JDBCConnectionPoolMBean`	A JDBC connection pool. See `JDBCConnectionPoolMBean` Javadoc.
`JDBCMultiPool`	A JDBC multipool, which is a pool of JDBC connection pools. See `JDBCMultiPool` Javadoc.
`JDBCDataSource`	A non-transactional data source. See `JDBCDataSource` Javadoc.
`JDBCTxDataSource`	A transactional data source. See `JDBCTxDataSource` Javadoc .

Figure 3-5 JDBC MBeans



JMS Configuration Namespace

WebLogic Server uses several MBeans to provide a management interface for its JMS services. Table 3-7 introduces the MBeans and Figure 3-6 illustrates the namespace in the sample MedRec domain.

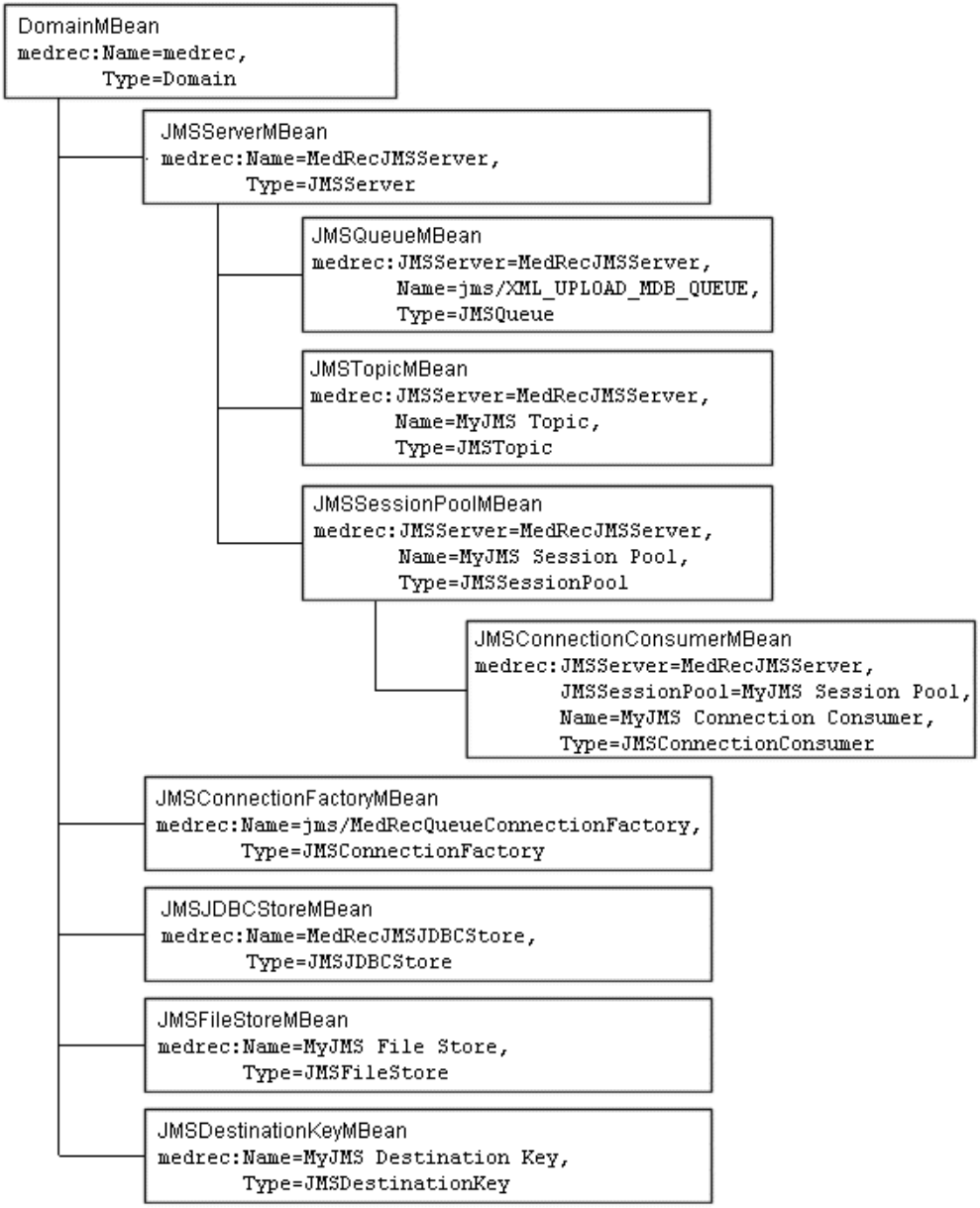
Table 3-7 MBeans for JMS

This MBean...	Configures...
`JMSServerMBean`	A JMS server. See `JMSMBean` Javadoc.
`JMSQueueMBean`	A JMS queue (Point-To-Point) destination for a JMS server. See `JMSQueueMBean` Javadoc.
`JMSTopicMBean`	A JMS topic (Pub/Sub) destination for a JMS server. See `JMSTopicMBean` Javadoc.

Table 3-7 MBeans for JMS

This MBean...	Configures...
`JMSSessionPoolMBean`	A server-managed pool of server sessions that enables an application to process messages concurrently. See `JMSSessionPoolMBean` Javadoc.
`JMSConnectionConsumerMBean`	A JMS connection consumer, which is a JMS destination (queue or topic) that retrieves server sessions and processes messages. See `JMSConnectionConsumerMBean` Javadoc.
`JMSConnectionFactoryMBean`	A JMS connection factory, which enables JMS clients to create JMS connections. See `JMSConnectionFactoryMBean` Javadoc.
`JMSJDBCStoreMBean`	A JMS JDBC store for storing persistent messages and durable subscribers in a JDBC-accessible database. See `JMSJDBCStoreMBean` Javadoc.
`JMSFileStoreMBean`	A disk-based JMS file store that stores persistent messages and durable subscribers in a file-system directory. See `JMSFileStoreMBean` Javadoc.
`JMSDestinationKeyMBean`	A key value for a destination, which is used to define the sort order of messages as they arrive on a destination. See `JMSDestinationKeyMBean` Javadoc.

Figure 3-6 JMS MBeans



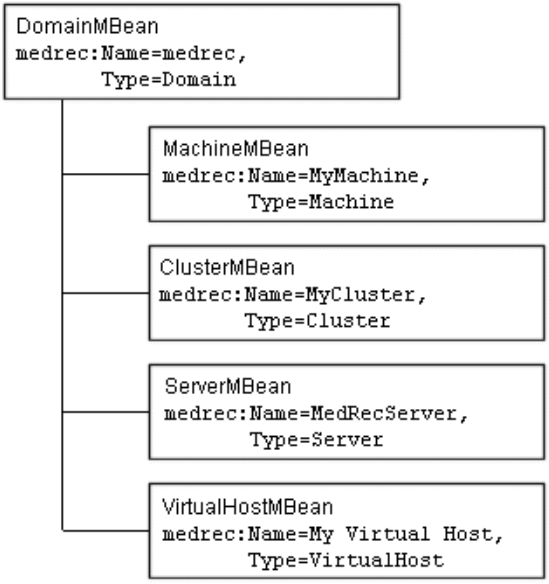
Clusters Configuration Namespace

WebLogic Server uses several MBeans to provide a management interface for clusters and cluster-related resources. Table 3-8 introduces the MBeans and Figure 3-7 illustrates the namespace in the sample MedRec domain.

Table 3-8 MBeans for Clusters

This MBean...	Configures...
`MachineMBean`	A representation of the WebLogic Server host on which a cluster member runs. Clusters use machines to determine default failover behavior. See `MachineMBean` Javadoc.
`ClusterMBean`	Cluster address and multicast settings. See `ClusterMBean` Javadoc.
`ServerMBean`	An individual server instance. See `ServerMBean` Javadoc.
`VirtualHostMBean`	Host names to which the cluster responds. See `VirtualHostMBean` Javadoc.

Figure 3-7 Cluster MBeans



Machines and Node Manager Configuration Namespace

If your domain consists of multiple server instances running on multiple WebLogic Server host computers, you can use machines and Node Manager to facilitate managing the life cycle of servers. WebLogic Server uses several MBeans to provide a management interface for machines and Node Managers. Table 3-9 introduces the MBeans and Figure 3-8 illustrates the namespace in the sample MedRec domain.

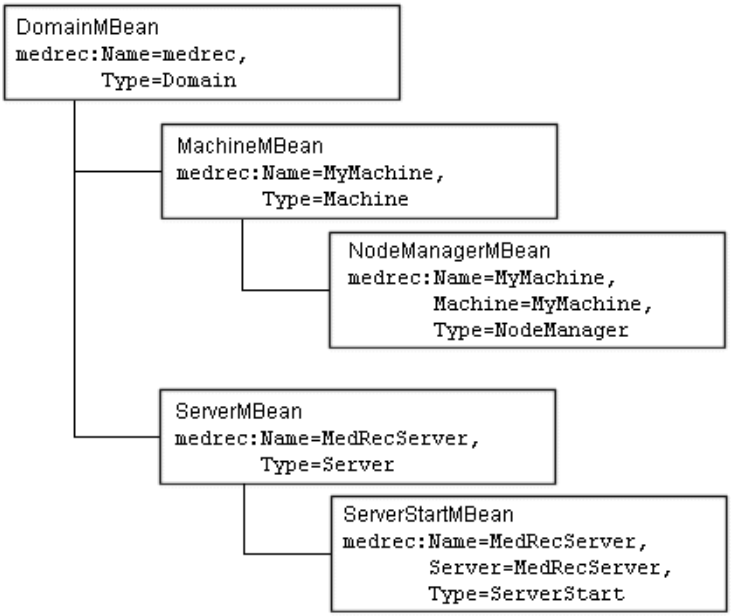
Table 3-9 MBeans for Machines and Node Manager

This MBean...	Configures...
`MachineMBean`	A representation of the WebLogic Server host on which a server instance runs. See `MachineMBean` Javadoc.

Table 3-9 MBeans for Machines and Node Manager

This MBean...	Configures...
`NodeManagerMBean`	Listen address, listen port, and security information that a server instance uses to communicate with a Node Manager running on a specific machine. See `NodeManagerMBean` Javadoc.
`ServerStartMBean`	Information that a Node Manager uses to start a Managed Server instance. See `ServerStartMBean` Javadoc.

Figure 3-8 Machines and Node Manager MBeans



Using weblogic.Admin to Find the WebLogicObjectName

If you are unsure which values to supply for an MBean's `WebLogicObjectName`, you can use the `weblogic.Admin` utility to find the `WebLogicObjectName`. The utility can return information only for WebLogic Server MBeans that are on an active server instance.

For example, to find the `WebLogicObjectName` for the Administration instance of the `LogMBean` in the `medrec` domain, enter the following command on the `MedRecServer` Administration Server, where the Administration Server is listening on port 8001 and `weblogic` is the name and password of a user who has permission to view MBean attributes:

```
java weblogic.Admin -url localhost:8001 -username weblogic -password
weblogic GET -pretty -type Log
```

The command returns the output in Listing 3-1. Notice that the command returns two MBeans of type `Log` on the Administration Server. The first MBean, `medrec:Name=MedRecServer,Server=MedRecServer,Type=Log`, has a child relationship with the `ServerMBean` of `MedRecServer`; this relationship indicates that the MBean is the `LogMBean` that configures the server-specific log file. The second MBean, `medrec:Name=medrec,Type=Log`, has no child relationship, which indicates that it configures the domain-wide log file.

The `-pretty` causes the `weblogic.Admin` utility to place each MBean attribute and value on a separate line. Without this argument, the utility surrounds each attribute/value pair with curly braces `{}`, but all output is on a single line.

Listing 3-1 Output from weblogic.Admin

```
---------------------------
MBeanName: "medrec:Name=MedRecServer,Server=MedRecServer,Type=Log"
        CachingDisabled: true
        FileCount: 7
        FileMinSize: 500
        FileName: MedRecServer\MedRecServer.log
        FileTimeSpan: 24
        Name: MedRecServer
        Notes:
        NumberOfFilesLimited: false
        ObjectName: MedRecServer
        Registered: false
        RotationTime: 00:00
        RotationType: none
        Type: Log
---------------------------
MBeanName: "medrec:Name=medrec,Type=Log"
```

```
        CachingDisabled: true
        FileCount: 7
        FileMinSize: 500
        FileName: ./logs/wl-domain.log
        FileTimeSpan: 24
        Name: medrec
        Notes:
        NumberOfFilesLimited: false
        ObjectName: medrec
        Registered: false
        RotationTime: 00:00
        RotationType: none
        Type: Log
```

To view the Local Configuration MBean instances of `LogMBean`, append `Config` to the value of the `type` argument:

```
java weblogic.Admin -url localhost:8001 -username weblogic -password
weblogic GET -pretty -type LogConfig
```

The command returns output in Listing 3-2. Notice that the `WebLogicObjectName` of the Local Configuration MBeans includes a `Location` component.

Listing 3-2 Local Configuration MBeans

```
---------------------------
MBeanName:
"medrec:Location=MedRecServer,Name=MedRecServer,ServerConfig=MedRecServer,
Type=LogConfig"
        CachingDisabled: true
        FileCount: 7
        FileMinSize: 500
        FileName: MedRecServer\MedRecServer.log
        FileTimeSpan: 24
        Name: MedRecServer
        Notes:
        NumberOfFilesLimited: false
```

```
        ObjectName: MedRecServer
        Registered: false
        RotationTime: 00:00
        RotationType: none
        Type: LogConfig
---------------------------
MBeanName: "medrec:Location=MedRecServer,Name=medrec,Type=LogConfig"
        CachingDisabled: true
        FileCount: 7
        FileMinSize: 500
        FileName: ./logs/wl-domain.log
        FileTimeSpan: 24
        Name: medrec
        Notes:
        NumberOfFilesLimited: false
        ObjectName: medrec
        Registered: false
        RotationTime: 00:00
        RotationType: none
        Type: LogConfig
```

Using weblogic.Admin to Find the Name of a Security Provider MBean

If you are unsure which values to supply for a security MBean's object name, you can use the `weblogic.Admin QUERY` command to retrieve the object name. The domain in which the security MBean exists must be active.

If you followed the recommended naming convention, or if you used the Administration Console to create the security MBean, you can use the following command to list all security MBeans in a domain:

```
java weblogic.Admin -url localhost:8001 -username weblogic -password
weblogic QUERY -pretty -pattern Security:*
```

Otherwise, you can use other forms of the `QUERY` command to find MBean names. See "QUERY" in *WebLogic Server Command Reference*.

CHAPTER 4

Accessing and Changing Configuration Information

Configuration MBeans on the Administration Server (Administration MBeans) configure the managed resources on all WebLogic Server instances in a domain. To enhance performance, each server instance creates and uses local replicas of the Administration MBeans. These local replicas are called Local Configuration MBeans.

Note: While you can view the values of Local Configuration MBeans, BEA recommends that you do not change attribute values in Local Configuration MBeans. Instead, change only the values of Administration MBean attributes. When the Managed Server replicates the data of other Managed Servers, it uses the values that are stored in Administration MBeans. Communication problems can occur if the values in Administration MBeans and Local Configuration MBeans differ.

The following sections provide examples for programmatically viewing and modifying the configuration of WebLogic Server resources using the `weblogic.Admin` utility, the JMX `MBeanServer` APIs, and the WebLogic Server type-safe interface:

Example: Using weblogic.Admin to View the Message Level for Standard Out

This example uses the `weblogic.Admin` utility to connect directly to a Managed Server and look up the value of its `StdoutSeverityLevel` attribute. This attribute, which belongs to the server's `ServerMBean`, specifies a threshold for determining which severity-level of messages a server prints to its standard out.

While BEA recommends that you use only Administration MBeans to change values, there might be situations in which it is preferable to look up the values that are in Local Configuration MBeans. For example, the Administration Server might be down, making it impossible for you to access Administration MBeans.

The example command:

1. Uses the `-url` argument to connect to a Managed Server that runs on a host named myHost and that listens on port 8001.
2. Uses the `-username` and `-password` arguments to specify the credentials of a user who has permission to view MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.
3. Uses the `GET` command to retrieve a Local Configuration MBean.

To specify a Local Configuration MBean, it removes `MBean` and appends `Config` to the `ServerMBean` interface name. Note that the `-type` value for a Local Configuration instance of the `ServerMBean` is `ServerConfig` while the `-type` value for the corresponding Administration MBean instance is `Server`. For more information, refer to the description of `Type` in Table 3-1, "WebLogic Server MBean Naming Conventions," on page 3-3.

Listing 4-1 Configuring the Message Level

```
java weblogic.Admin -url myHost:8001 -username weblogic -password weblogic
GET -pretty -type ServerConfig

----------------------------
MBeanName: "medrec:Location=MedRecServer,Name=MedRecServer,Type=ServerConfig"
        AcceptBacklog: 50
        AdministrationPort: 0
...

        StdoutDebugEnabled: false
        StdoutEnabled: true
        StdoutFormat: standard
        StdoutLogStack: true
        StdoutSeverityLevel: 16
```

Example: Configuring the Message Level for Standard Out

The class in this example changes the value of the `StdoutSeverityLevel` attribute in the `weblogic.management.configuration.ServerMBean` to change the level of messages that a server instance named `MedRecServer` sends to standard out.

Because the example is changing configuration values, it changes the value in the Administration MBean and relies on the WebLogic management services to propagate the change to the Managed Server.

The example class:

1. Uses JNDI to look up the Administration `MBeanHome` interface on the Administration Server.
2. Uses the `MBeanHome.getMBean(String name, String type)` API to retrieve the type-safe interface of the `ServerMBean` Administration MBean for a server instance named Server1.

3. Uses the type-safe interface to invoke the `ServerMBean.setStdoutSeverityLevel` method and set the severity level to 64.

In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 4-2 Configuring Standard Out Severity Level

```
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.management.MBeanServer;
import javax.management.Attribute;
import java.lang.Object;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.configuration.ServerMBean;

public class ChangeStandardOut1 {

    public static void main(String[] args) {
        MBeanHome home = null;
        ServerMBean server = null;
        //domain variables
        String url = "t3://localhost:7001";
        String username = "weblogic";
        String password = "weblogic";
        String serverName = "Server1";

        //setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //getting the Administration MBeanHome
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);

            // Using MBeanHome.getMBean(name, type) to retrieve a type-safe
            // interface for a ServerMBean
            server = (ServerMBean)home.getMBean(serverName,"Server");
```

```
            // Using ServerMBean.setStdoutSeverityLevel
            server.setStdoutSeverityLevel(64);

            //  Providing feedback that operation succeeded.
            System.out.println("Changed standard out severity level to: " +
                               server.getStdoutSeverityLevel());
        } catch (Exception e) {
            System.out.println("Caught exception: " + e);
        }
    }
}
```

Setting and Getting Encrypted Values

To prevent unauthorized access to sensitive data such as passwords, some attributes in configuration MBeans are encrypted. The attributes persist their values in the domain's `config.xml` file as an encrypted string and represent the in-memory value in the form of an encrypted byte array. The names of encrypted attributes end with `Encrypted`. For example, the `JDBCConnectionPoolMBean` exposes the password that is used to access the database in an attribute named `PasswordEncrypted`.

The following sections describe how to work with encrypted attributes:

- "Set the Value of an Encrypted Attribute" on page 4-5
- "Compare an Unencrypted Value with an Encrypted Value" on page 4-6
- "Example: Setting and Getting an Encrypted Attribute" on page 4-6

Set the Value of an Encrypted Attribute

To set the value of an encrypted attribute, encode a `String` object as a byte array and pass the output directly to the setter method as a parameter. Do not assign the byte array to a variable because this causes the unencrypted byte array to remain in memory until garbage collection removes it.

For example, if you use `weblogic.management.MBeanHome`:

```
ServerMBean.setCustomIdentityKeyStorePassPhraseEncrypted(
    (new String("myNewCustomIdentityKeyStorePassPhrase")).getBytes());
```

If you use `weblogic.management.RemoteMBeanServer`:

```
Attribute passphrase = new Attribute("CustomIdentityKeyStorePassPhrase",
    new String("myNewCustomIdentityKeyStorePassPhrase").getBytes());

String server = "examples:Name=examplesServer,Type=Server";
ObjectName serverOn = new ObjectName(server);
RemoteMBeanServer.setAttribute(serverOn, passphrase);
```

Compare an Unencrypted Value with an Encrypted Value

A management application might need to compare a password or some other value that a user enters with a value that is in an MBean's encrypted attribute. Instead of decrypting the MBean attribute value and risk exposing the data to someone with unauthorized access, you encrypt the user-supplied value and compare the two encrypted values.

You must encrypt the user-supplied value on the same server that originally encrypted the MBean value. Each server uses its own salt file to encrypt data unless the server is sharing its root directory with another server. See "A Server's Root Directory" in *Configuring and Managing WebLogic Server*.

To compare a password or some other value that a user enters with a value that is in an encrypted attribute:

1. On the same server that set the encrypted value in the MBean, write the user-supplied value as a byte array and pass the byte array to the `weblogic.management.EncryptionHelper.encrypt()` method.
2. Use the getter method of the MBean's encrypted value to retrieve its encrypted byte array.

   For example, invoke `JDBCConnectionPoolMBean.getPasswordEncrypted`, which returns an encrypted byte array.
3. Compare the two encrypted byte arrays.

Example: Setting and Getting an Encrypted Attribute

The class in Listing 4-3 retrieves and displays the encrypted pass phrase for a custom identity key store. Then it changes the pass phrase, retrieves and displays the newly encrypted phrase.

Because the example is changing configuration values, it changes the value in the Administration MBean.

The example class:

1. Uses JNDI to look up the Administration `MBeanHome` interface on the Administration Server.

2. Uses the `MBeanHome.getMBean(String name, String type)` API to retrieve the type-safe interface of the `ServerMBean` Administration MBean for a server instance named myserver.
3. Gets the encrypted pass phrase for the custom identity key store by invoking `ServerMBean.getCustomIdentityKeyStorePassPhraseEncrypted()`.

   The `getCustomIdentityKeyStorePassPhraseEncrypted()` method returns an encrypted byte array.
4. Displays the encrypted value by converting the encrypted byte array to a `String` object and printing the object to standard out.
5. Sets a new pass phrase for the custom identity key store by doing the following:
   a. Creates a `String` object that contains the new pass phrase.
   b. Uses `String.getBytes()` to create a byte array that contains the value of the `String` object.
   c. Passes the byte array as input for the `ServerMBean.setCustomIdentityKeyStorePassPhraseEncrypted` method:

```
mbean.setCustomIdentityKeyStorePassPhraseEncrypted((new
String("myCustomIdentityKeyStorePassPhrase")).getBytes())
```

6. Gets the encrypted pass phrase for the custom identity key store by invoking `ServerMBean.getCustomIdentityKeyStorePassPhraseEncrypted()`.
7. Displays the unencrypted value by converting the unencrypted byte array to a `String` object and printing the object to standard out.

Listing 4-3 Getting and Setting Encrypted Values

```
import java.util.*;
import java.rmi.RemoteException;
import javax.naming.*;
import javax.management.MBeanServer;
import javax.management.Attribute;
import javax.management.InstanceNotFoundException;

import weblogic.jndi.Environment;
import weblogic.management.WebLogicMBean;
```

```
import weblogic.management.MBeanHome;
import weblogic.management.configuration.ServerMBean;

public class GetSetEncrypted {
    private static MBeanHome home = null;
    static MBeanHome getHome(String[] args) {
       Context ctx= null;
       Hashtable ht = new Hashtable();
       ht.put(Context.INITIAL_CONTEXT_FACTORY,
           "weblogic.jndi.WLInitialContextFactory");
       ht.put(Context.PROVIDER_URL, "t3://localhost:7001");
       ht.put(Context.SECURITY_PRINCIPAL, args[0]);
       ht.put(Context.SECURITY_CREDENTIALS, args[1]);
       try {
          System.out.println("Getting the initialContext ...");
          ctx = new InitialContext(ht);
          System.out.println("Got initialContext");
          home = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
        } catch(Exception e) {
            e.printStackTrace();
       }
       return home;
    }

    static void getsetServerMBean() throws Exception {
       byte[] bytes = null;
       String serverName = "myserver";
       ServerMBean mbean=(ServerMBean)home.getMBean(serverName,"Server");
       System.out.println("Found admin mbean,name=" +
           ((WebLogicMBean)mbean).getObjectName());
       bytes = mbean.getCustomIdentityKeyStorePassPhraseEncrypted();
       if (bytes != null) {
          System.out.println("\n\ngetCustomIdentityKeyStorePassPhraseEncry
                  pted returned=\n" + (new String(bytes)));
       } else {
          System.out.println("\n\ngetEncrypted Attribute returned NULL");
       }
       System.out.println("\n\nInvoking
            setCustomIdentityKeyStorePassPhraseEncrypted() with
```

```
            myNewCustomIdentityKeyStorePassPhrase");
        mbean.setCustomIdentityKeyStorePassPhraseEncrypted((new
           String("myNewCustomIdentityKeyStorePassPhrase")).getBytes());
        bytes = mbean.getCustomIdentityKeyStorePassPhraseEncrypted();
        System.out.println("\n\nAfter
            setCustomIdentityKeyStorePassPhraseEncrypted(),
            getCustomIdentityKeyStorePassPhraseEncrypted returned=\n" +
            (new String(bytes)));
    }

    public static void main (String[] args) {
        getHome(args);
        try {
           getsetServerMBean();
        } catch (Exception e) {
           e.printStackTrace();
        }
    }
}
```

CHAPTER 5

Accessing Runtime Information

WebLogic Server includes a large number of MBeans that provide information about the runtime state of managed resources. If you want to create applications that view and modify this runtime data, you must first use the `MBeanServer` interface or the WebLogic Server type-safe interface to retrieve Runtime MBeans. Then you use APIs in the `weblogic.management.runtime` package to view or change the runtime data. For information about viewing the API documentation, refer to "Documentation for Runtime MBean APIs" on page 1-11.

The following sections provide examples for retrieving and modifying runtime information about WebLogic Server domains and server instances:

Example: Determining the Active Domain and Servers

The `MBeanHome` interface includes APIs that you can use to determine the name of the currently active domain and the name of server instances.

The example class in Listing 5-1 gets the name of the current domain and the names of all active servers in the domain:

1. Retrieves the Administration `MBeanHome` interface.

   Note: To get only the name of the current server instance, use the Local `MBeanHome` interface instead of Administration `MBeanHome`. See "Getting the Name of the Current Server Instance" on page 5-4.

2. Uses `MBeanHome.getActiveDomain().getName()` to retrieve the name of the domain.
3. Uses the `getMBeansByType` method to retrieve the set of all `ServerRuntime` MBeans in the domain.
4. Iterates through the set and compares the names of the `ServerRuntimeMBean` instances with the name of the WebLogic Server instance.
5. Invokes the `serverRuntime.getState` method to retrieve the state of each server instance.
6. Compares the value that the `getState` method returns with a field name from the `weblogic.management.runtime.ServerStates` class. This class contains one field for each state within a server's life cycle. For more information about `weblogic.management.runtime.ServerStates`, refer to the WebLogic Server Javadoc.
7. If the instance is active, it prints the name of the server.

In the following example, `weblogic` is the username and password for a user who has permission to view and modify MBean attributes. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-1 Determining the Active Domain and Servers

```
import java.util.Set;
import java.util.Iterator;
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServerRuntimeMBean;
import weblogic.management.runtime.ServerStates;

public class getActiveDomainAndServers {
    public static void main(String[] args) {
        MBeanHome home = null;

        //url of the Administration Server
        String url = "t3://localhost:7001";
```

```
        String username = "weblogic";
        String password = "weblogic";

        //setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //getting the Administration MBeanHome
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        //getting the name of the active domain
        try {
            System.out.println("Active Domain: " +
                     home.getActiveDomain().getName() );
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        }

        //getting the names of servers in the domain
        System.out.println("Active Servers: ");
        Set mbeanSet = home.getMBeansByType("ServerRuntime");
        Iterator mbeanIterator = mbeanSet.iterator();
        while(mbeanIterator.hasNext()) {
            ServerRuntimeMBean serverRuntime =
                (ServerRuntimeMBean)mbeanIterator.next();
            //printing the names of active servers
            if(serverRuntime.getState().equals(ServerStates.RUNNING)){
                System.out.println("Name: " + serverRuntime.getName());
                System.out.println("ListenAddress: " +
                          serverRuntime.getListenAddress());
                 System.out.println("ListenPort: " +
                          serverRuntime.getListenPort());
                 //count++;
            }
        }

        System.out.println("Number of servers active in the domain: " +
                             mbeanSet.size());
    }
}
```

Getting the Name of the Current Server Instance

To retrieve only the name of the current server instance, create a JMX client that does the following:

1. Retrieves the Local `MBeanHome` interface.
2. Uses `MBeanHome.getMBeansByType()` to retrieve the set of all `ServerRuntime` MBeans in the server.

   Because the Local `MBeanHome` interface can access only the runtime MBeans for the current server instance, the `getMBeansByType()` method returns a set that contains only the `ServerRuntimeMBean` for the current server.
3. Invokes the `ServerRuntimeMBean.getName` method.

Listing 5-2 is a code segment that you can use for a JMX client that runs within a WebLogic Server JVM.

Listing 5-2

```
// Get a JNDI Context
weblogic.jndi.Environment env = new Environment();
env.setSecurityPrincipal(USERNAME);
env.setSecurityCredentials(PASSWORD);
Context ctx = env.getInitialContext();

// Get the Local MBeanHome
MBeanHome home =
   (MBeanHome)ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);

Set s = home.getMBeansByType("ServerRuntime");
Iterator i = s.iterator();
ServerRuntimeMBean serverRT = (ServerRuntimeMBean) i.next();
String serverName = serverRT.getName();
return serverName;
```

Using weblogic.Admin to Determine Active Domains and Servers

While you can compile and run the example code in Listing 5-1 to determine active domains and servers, you can use the `weblogic.Admin` utility to accomplish a similar task without having to compile Java classes.

The following command returns the name of the currently active domain, where `AdminServer` is the domain's Administration Server, `MyHost` is the Administration Server's host computer, and `weblogic` is the name and password of a user who has permission to view MBean attributes:

```
java weblogic.Admin -url MyHost:8001 -username weblogic -password weblogic
GET -type DomainRuntime -property Name
```

The command output includes the `WebLogicObjectName` of the `DomainRuntimeMBean` and the value of its `Name` attribute:

```
{MBeanName="myDomain:Location=AdminServer,Name=myDomain,ServerRuntime=Admi
nServer,Type=DomainRuntime"{Name=myDomain}}
```

To see a list of all server instances that are currently active, you use ask the Administration Server to retrieve all `ServerRuntime` MBeans that are registered in its Administration `MBeanHome` interface. (Only active server instances register `ServerRuntime` MBeans with the Administration `MBeanHome` interface.)

You must specify the `-adminurl` argument to instruct the `GET` command to use the Administration Server's Administration `MBeanHome` interface:

```
java weblogic.Admin -adminurl MyHost:8001 -username weblogic -password
weblogic GET -type ServerRuntime -property State
```

The command output includes the `WebLogicObjectName` of all `ServerRuntime` MBeans and the value of each `State` attribute:

```
---------------------------
MBeanName: "myDomain:Location=MedRecMS2,Name=MedRecMS2,Type=ServerRuntime"
        State: RUNNING

---------------------------
MBeanName:
"myDomain:Location=AdminServer,Name=AdminServer,Type=ServerRuntime"
        State: RUNNING
```

```
----------------------------
MBeanName: "myDomain:Location=MedRecMS1,Name=MedRecMS1,Type=ServerRuntime"
        State: RUNNING
```

Example: Viewing and Changing the Runtime State of a WebLogic Server Instance

The `weblogic.management.runtime.ServerRuntimeMBean` interface provides runtime information about a WebLogic Server instance. For example, it indicates which listen ports and addresses a server is using. It also includes operations that can gracefully or forcefully shut down a server.

This section provides examples of finding `ServerRuntimeMBean` and using it to gracefully shut down a server instance. For more information about graceful shutdowns and controlling the graceful shutdown period, refer to "Graceful Shutdown" in *Configuring and Managing WebLogic Server*.

Each example illustrates a different way of retrieving `ServerRuntimeMBean`:

- "Using a Local MBeanHome and getRuntimeMBean()" on page 5-6
- "Using the Administration MBeanHome and getMBean()" on page 5-10
- "Using the Administration MBeanHome and getMBeansByType()" on page 5-8
- "Using the MBeanServer Interface" on page 5-12

You cannot use the `weblogic.Admin` utility to change the value of Runtime MBean attributes.

Using a Local MBeanHome and getRuntimeMBean()

Each WebLogic Server instance hosts its own `MBeanHome` interface, which provides access to the Local Configuration and Runtime MBeans on the server instance. As opposed to using the Administration `MBeanHome` interface, using the local `MBeanHome` saves you the trouble of filtering Runtime MBeans to find those that apply to the current server. It also uses fewer network hops to access MBeans, because you are connecting directly to the server (instead of routing requests through the Administration Server).

The `MBeanHome` interface includes the `getRuntimeMBean()` method, which returns only Runtime MBeans that reside on the current WebLogic Server. If you invoke `MBeanHome.getRuntimeMBean()` on the Administration Server, it returns only the Runtime MBeans that manage and monitor the Administration Server.

The example class in Listing 5-3 does the following:

1. Configures a `javax.naming.Context` object with information for connecting to a server instance that listens for requests at `t3://ServerHost:7001`.
2. Uses the `Context.lookup` method to retrieve the local `MBeanHome` interface for the server instance.

   The `MBeanHome.LOCAL_JNDI_NAME` field returns the JNDI name of the current server's local `MBeanHome`.
3. Uses the `MBeanHome.getRuntimeMBean(String name,String type)` method to retrieve the `ServerRuntimeMBean` for the current server instance.
4. Invokes `ServerRuntimeMBean` methods to retrieve and modify the server state.

In the following example, `weblogic` is the username and password for a user who has permission to view and modify MBean attributes and `Server1` is the name of the WebLogic Server instance for which you want to view and change status. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-3 Using a Local MBeanHome and getRuntimeMBean()

```
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServerRuntimeMBean;

public class serverRuntime1 {

    public static void main(String[] args) {
        MBeanHome home = null;

        //domain variables
        String url = "t3://ServerHost:7001";
        String serverName = "Server1";
        String username = "weblogic";
        String password = "weblogic";
        ServerRuntimeMBean serverRuntime = null;

        //setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
```

```
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

           //getting the local MBeanHome
            home = (MBeanHome) ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);
            System.out.println("Got the MBeanHome: " + home + " for server: " +
                                                                  serverName);
        } catch (Exception e) {
            System.out.println("Caught exception: " + e);
        }

        // Here we use the getRuntimeMBean method to access the
       //ServerRuntimeMbean of the server instance.

        try {
            serverRuntime =
                          (ServerRuntimeMBean)home.getRuntimeMBean
                          (serverName,"ServerRuntime");
            System.out.println("Got serverRuntimeMBean: " + serverRuntime);

            //Using ServerRuntimeMBean to retrieve and change state.
            System.out.println("Current state: " + serverRuntime.getState() );
            serverRuntime.shutdown();
            System.out.println("Current state: " + serverRuntime.getState() );
        } catch (javax.management.InstanceNotFoundException e) {
            System.out.println("Caught exception: " + e);
        }
    }
}
```

Using the Administration MBeanHome and getMBeansByType()

Like the example in Listing 5-1, "Determining the Active Domain and Servers," on page 5-2, the example class in this section uses the Administration `MBeanHome` interface to retrieve a `ServerRuntime` MBean. The Administration `MBeanHome` provides a single access point for all MBeans in the domain, but it requires you to either construct the `WebLogicObjectName` of the MBean you want to retrieve or to filter MBeans to find those that apply to a specific current server.

The example class in Listing 5-4 does the following:

1. Retrieves the Administration `MBeanHome` interface.

2. Uses the `MBeanHome.getMBeansByType` method to retrieve the set of all `ServerRuntime` MBeans in the domain.
3. Assigns the list of MBeans to a `Set` object and uses methods of the `Set` and `Iterator` interfaces to iterate through the list.
4. Uses the `ServerRuntimeMBean.getName` method to retrieve the `Name` component of the MBean's `WebLogicObjectName`. It then compares the `Name` value with another value.
5. When it finds the `ServerRuntimeMBean` for a specific server instance, it uses the `ServerRuntimeMBean.getState` method to return the current server state.
6. Then it invokes the `ServerRuntimeMBean.shutdown()` method, which initiates a graceful shutdown.

In the following example, `weblogic` is the username and password for a user who has permission to change the state of servers, and `Server1` is the name of the WebLogic Server instance for which you want to view and change status. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-4 Using the Administration MBeanHome and getMBeansByType()

```
import java.util.Set;
import java.util.Iterator;
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServerRuntimeMBean;

public class serverRuntimeInfo {
    public static void main(String[] args) {
        MBeanHome home = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "Server1";
        String username = "weblogic";
        String password = "weblogic";
        ServerRuntimeMBean serverRuntime = null;
        Set mbeanSet = null;
        Iterator mbeanIterator = null;

        //Setting the initial context
        try {
            Environment env = new Environment();
```

```
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            // Getting the Administration MBeanHome.
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Admin MBeanHome: " + home );
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        // Using the getMBeansByType method to get the set of
        //ServerRuntime mbeans.
        try {
            mbeanSet = home.getMBeansByType("ServerRuntime");
            mbeanIterator = mbeanSet.iterator();
            // Comparing the name of the server in each ServerRutime
            // MBean to the value specified by serverName
            while(mbeanIterator.hasNext()) {
                serverRuntime = (ServerRuntimeMBean)mbeanIterator.next();
                if(serverRuntime.getName().equals(serverName)) {
                    System.out.println("Found the serverRuntimembean: " +
                                       serverRuntime + " for: " + serverName);
                    System.out.println("Current state: " +
                                                 serverRuntime.getState() );
                    System.out.println("Stopping the server ...");
                    serverRuntime.shutdown();
                    System.out.println("Current state: " +
                                        serverRuntime.getState() );
                }
            }
        } catch (Exception e) {
            System.out.println("Caught exception: " + e);
        }
    }
}
```

Using the Administration MBeanHome and getMBean()

Instead of retrieving a list of all MBeans and then filtering the list to find the `ServerRuntimeMBean` for a specific server, this example uses the MBean naming conventions to construct the `WebLogicObjectName` for the `ServerRuntimeMBean` on a server instance named `Server1`. For information about constructing a `WebLogicObjectName`, refer to Table 3-1, “WebLogic Server MBean Naming Conventions,” on page 3-3.

To make sure that you supply the correct object name, use the `weblogic.Admin GET` command. For example, the following command returns the object name and list of attributes of the `ServerRuntimeMBean` for a server instance that runs on a host computer named `MyHost`:

```
java weblogic.Admin -url http://MyHost:7001 -username weblogic
 -password weblogic GET -pretty -type ServerRuntime
```

For more information about using the `weblogic.Admin` utility to find information about MBeans, refer to "Commands for Managing WebLogic Server MBeans" in the *WebLogic Server Command Line Reference*.

In Listing 5-5, `weblogic` is the username and password for a user who has permission to view and modify MBean attributes, `Server1` is the name of the WebLogic Server instance for which you want to view and change status, and `mihirDomain` is the name of the WebLogic Server administration domain. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-5 Using the Administration MBeanHome and getMBean()

```
import java.util.Set;
import java.util.Iterator;
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServerRuntimeMBean;
import weblogic.management.WebLogicObjectName;

public class serverRuntimeInfo2 {
    public static void main(String[] args) {
        MBeanHome home = null;
        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "Server1";
        String username = "weblogic";
        String password = "weblogic";
        String domain = "medrec";
        ServerRuntimeMBean serverRuntime = null;

        //setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
```

```
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got Admin MBeanHome from the Admin server: "
                               + home);
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        try {
            WebLogicObjectName objName = new WebLogicObjectName(serverName,
                          "ServerRuntime",home.getDomainName(),serverName);
           System.out.println("Created WebLogicObjectName: " + objName);
            serverRuntime = (ServerRuntimeMBean)home.getMBean(objName);
            System.out.println("Got the serverRuntime using the adminHome: " +
                               serverRuntime );
            System.out.println("Current state: " + serverRuntime.getState() );
            System.out.println("Stopping the server ...");

            //changing the state to SHUTDOWN
            serverRuntime.shutdown();
            System.out.println("Current state: " + serverRuntime.getState() );
        } catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

Using the MBeanServer Interface

The example in this section uses a standard JMX approach for interacting with MBeans. It uses the Administration `MBeanHome` interface to retrieve the `javax.management.MBeanServer` interface and then uses `MBeanServer` to retrieve the value of the `ListenPort` attribute of the `ServerRuntimeMBean` for a server instance named `Server1`.

In the following example, `weblogic` is the username and password for a user who has permission to view and modify MBean attributes and `mihirDomain` is the name of the WebLogic Server administration domain. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-6 Using the Administration MBeanHome and getMBean()

```
import java.util.Set;
import java.util.Iterator;
import javax.naming.Context;
import javax.management.MBeanServer;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicObjectName;

public class serverRuntimeInfo2 {
    public static void main(String[] args) {
        MBeanHome home = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "MedRecServer";
        String username = "weblogic";
        String password = "weblogic";
        Object attributeValue = null;
        MBeanServer homeServer = null;

        //setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            // Getting the Administration MBeanHome.
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got Admin MBeanHome from the Admin server: " +
                                                                    home);
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }
        try {
            // Creating the mbean object name.
            WebLogicObjectName objName = new WebLogicObjectName(serverName,
                        "ServerRuntime",home.getDomainName(),serverName);
            System.out.println("Created WebLogicObjectName: " + objName);

            //Retrieving the MBeanServer interface
            homeServer = home.getMBeanServer();

            //Retrieving the ListenPort attribute of ServerRuntimeMBean
            attributeValue = homeServer.getAttribute(objName, "ListenPort");
```

```
            System.out.println("ListenPort for " + serverName + " is:" +
                                                                attributeValue);
        } catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

Example: Viewing Runtime Information About Clusters

The example in this section retrieves the number and names of WebLogic Server instances currently running in a cluster. It uses `weblogic.management.runtime.ClusterRuntimeMBean`, which provides information about a single Managed Server's view of the members of a WebLogic cluster.

Only Managed Servers host instances of `ClusterRuntimeMBean`, and you must retrieve the `ClusterRuntimeMBean` instance from a Managed Server that is actively participating in a cluster.

To make sure that it retrieves a `ClusterRuntimeMBean` from an active Managed Server that is in a cluster, this example does the following:

1. Retrieves the Administration `MBeanHome`, which runs on the Administration Server and can provide access to all `ClusterRuntimeMBeans` in the domain.
2. Retrieves all `ClusterRuntimeMBeans` and determines whether they belong to a specific cluster.
3. Finds one `ClusterRuntimeMBean` for a Managed Server in the cluster of interest.
4. Uses the `ClusterRuntimeMBean` APIs on the Managed Server to determine the number and name of active servers in the cluster.

In the example, `weblogic` is the username and password for a user who has permission to view and modify MBean attributes. For information about permissions to modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 5-7 Retrieving a List of Servers Running in a Cluster

```
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
```

```
import javax.naming.Context;
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import javax.management.ObjectName;
import weblogic.management.WebLogicMBean;
import weblogic.management.runtime.ClusterRuntimeMBean;
import weblogic.management.WebLogicObjectName;
import weblogic.management.MBeanHome;

public class getRunningServersInCluster {
    public static void main(String[] args) {
        MBeanHome home = null;

        //domain variables
        String url = "t3://localhost:7001"; //url of the Administration Server
        /* If you have more than one cluster in your domain, define a list of
         * all the servers in the cluster. You compare the servers in the domain
         * with this list to determine which servers are in a specific cluster.
         */
        String server1 = "cs1"; // name of server in the cluster
        String server2 = "cs2"; // name of server in the cluster
        String username = "weblogic";
        String password = "weblogic";
        ClusterRuntimeMBean clusterRuntime = null;
        Set mbeanSet = null;
        Iterator mbeanIterator = null;
        String name = "";
        String[] aliveServerArray = null;

        //Setting the initial context
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            // Getting the Administration MBeanHome.
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);

            // Retrieving a list of ClusterRuntime MBeans in the domain.
            mbeanSet = home.getMBeansByType("ClusterRuntime");
            mbeanIterator = mbeanSet.iterator();
            while(mbeanIterator.hasNext()) {

                // Retrieving one ClusterRuntime MBean from the list.
                clusterRuntime = (ClusterRuntimeMBean)mbeanIterator.next();
                // Getting the name of the ClusterRuntime MBean.
                name = clusterRuntime.getName();
                // Determining if the current ClusterRuntimeMBean belongs to a
```

```
                // server in the cluster of interest.
                if(name.equals(server1) || name.equals(server2) ) {
                    // Using the current ClusterRuntimeMBean to retrieve the
                    // number of servers in the cluster.
                    System.out.println("\nNumber of active servers in the
                         cluster: " + clusterRuntime.getAliveServerCount());
                    // Retrieving the names of servers in the cluster.
                    aliveServerArray = clusterRuntime.getServerNames();
                    break;
                }
            }
        } catch (Exception e) {
             System.out.println("Caught exception: " + e);
        }
        if(aliveServerArray == null) {
            System.out.println("\nThere are no running servers in the cluster");
            System.exit(1);
        }

        System.out.println("\nThe running servers in the cluster are: ");
        for (int i=0; i < aliveServerArray.length; i++) {
            System.out.println("server " + i + " : " + aliveServerArray[i]);
        }
    }
}
```

Viewing Runtime Information for EJBs

For each EJB that you deploy on a server instance, WebLogic Server instantiates MBean types from the `weblogic.management.runtime` package (see Table 5-1). For more information about the MBeans in the `weblogic.management.runtime` package, refer to the WebLogic Server Javadoc.

Table 5-1 MBeans that Provide Runtime Information for EJBs

MBean Type	Description
`EJBComponentRuntimeMBean`	The top level interface for all runtime information collected for an EJB module.
`StatefulEJBRuntimeMBean`	Instantiated for stateful session beans only. Contains methods for accessing EJB runtime information collected for a Stateful Session Bean.

Table 5-1 MBeans that Provide Runtime Information for EJBs

MBean Type	Description
`StatelessEJBRuntimeMBean`	Instantiated for stateless session beans only. Contains methods for accessing EJB runtime information collected for a Stateless Session Bean.
`MessageDrivenEJBRuntimeMBean`	Instantiated for message driven bean only. Contains methods for accessing EJB runtime information collected for a Message Driven Bean.
`EntityEJBRuntimeMBean`	Contains methods for accessing EJB runtime information collected for an Entity Bean.
`EJBCacheRuntimeMBean`	Contains methods for accessing cache runtime information collected for an EJB.
`EJBLockingRuntimeMBean`	Contains methods for accessing lock manager runtime information collected for an EJB.
`EJBTransactionRuntimeMBean`	Contains methods for accessing transaction runtime information collected for an EJB.
`EJBPoolRuntimeMBean`	Instantiated for stateless session beans only. Contains methods for accessing free pool runtime information collected for a stateless session EJB. WebLogic Server uses a free pool to improve performance and throughput for stateless session EJBs. The free pool stores unbound stateless session EJBs. Unbound EJB instances are instances of a stateless session EJB class that are not processing a method call.

WebLogic Server provides an additional, abstract interface, `EJBRuntimeMBean`, which contains methods that the other EJB runtime MBeans use.

EJB runtime MBeans are instantiated within a hierarchy. For example:

- Each EJB jar file exposes its runtime data through an instance of `EJBComponentRuntimeMBean`.
- Each entity bean within the jar exposes its runtime data through an instance of `EntityEJBRuntimeMBean`.

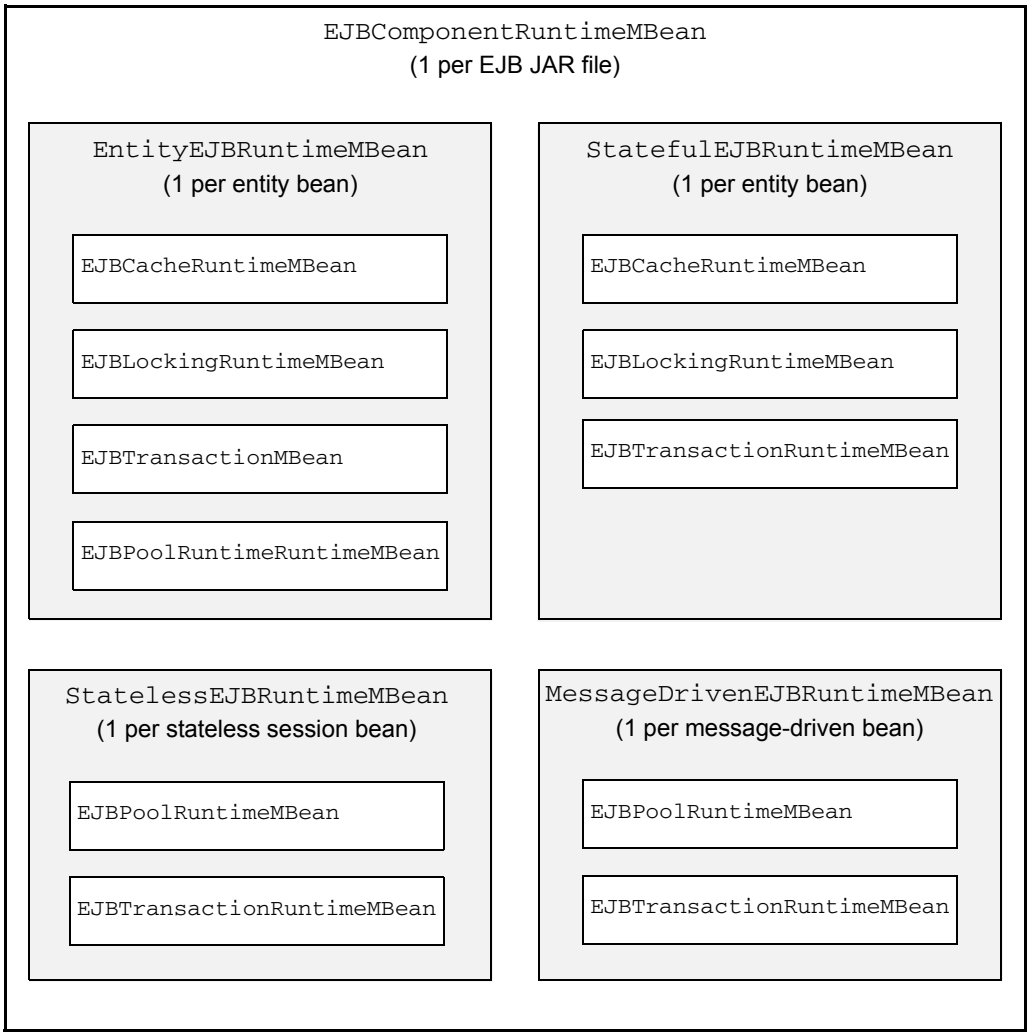
- Each `EntityEJBRuntimeMBean` is the parent of up to four additional MBeans.

  `EntityEJBRuntimeMBean` is always the parent of the `EJBCacheRuntimeMBean`, `EJBTransactionRuntimeMBean`, and `EJBPoolRuntimeMBean` MBeans. The fourth child MBean, `EJBLockingRuntimeMBean`, is only created if the entity bean uses an exclusive concurrency strategy (which is configured in the `weblogic-ejb-jar.xml` deployment descriptor).

Depending on the type of runtime data that you retrieve, you typically also need to retrieve the name of any parent MBeans to provide context for the data. For example, if you retrieve the value of `EJBTransactionRuntimeMBean.TransactionsRolledBackTotalCount`, you also retrieve the name of the parent `EJBEntityRuntimeMBean` to determine which entity bean the value comes from.

Figure 5-1 illustrates the hierarchical relationships.

Figure 5-1 Hierarchy of EJB Runtime MBeans



Example: Retrieving Runtime Information for All Stateful and Stateless EJBs

To retrieve runtime information for all EJBs deployed in a domain, the example in Listing 5-8 does the following:

1. Connects to the Administration Server and retrieves the Administration `MBeanHome` interface.

   If you want to retrieve runtime information only for the EJBs that are deployed on a specific server instance, you can connect to the specific server instance and retrieve the local `MBeanHome` interface. For more information, refer to "Example: Retrieving a Local MBeanHome from an Internal Client" on page 2-9.

2. To display the percentage of times a stateless bean instance wasn't available in the free pool when an attempt was made to obtain one, the example:

   a. Invokes the `MBeanHome.getMBeansByType` to retrieve all `StatelessEJBRuntime` MBeans.

   b. For each stateless EJB, it invokes the `displayEJBInfo` method (which is defined later in this class). This method:

   – Invokes the `StatelessEJBRuntimeMBean.getEJBName` method (which all EJB runtime MBeans inherit from `EJBRuntimeMBean`) to retrieve the name of the MBean.

   – Walks up the MBean hierarchy to retrieve the names of the parent EJB component and application.

     All EJBs are packaged within an EJB component, which functions as a J2EE module. EJB components can be packaged with an enterprise application.

   c. Invokes the `StatelessEJBRuntime.getPoolRuntime` method to retrieve the `EJBPoolRuntimeMBean` that is associated with the stateless EJB.

   d. Invokes the `EJBPoolRuntimeMBean.getMissTotalCount` method to retrieve the number of failed attempts.

3. To determine percentage of transactions that have been rolled back for each stateful EJB in the domain, the example:

   a. Invokes the `MBeanHome.getMBeansByType` to retrieve all `StatefulEJBRuntime` MBeans.

   b. Invokes the `displayEJBInfo` method (which is defined later in this class).

   c. Invokes the `EJBRuntime.getTransactionRuntime` method to retrieve the `EJBTransactionRuntimeMBean` that is associated with the stateful EJB.

   d. Invokes the `EJBTransactionRuntimeMBean.getTransactionsRolledBackTotalCount` and `getTransactionsCommittedTotalCount` methods.

e. Divides the number of committed transactions by the number rolled transactions to determine the percentage of rolled back transactions.

Listing 5-8 Viewing Runtime Information for EJBs

```
import java.util.Iterator;
import java.util.Set;

import javax.management.InstanceNotFoundException;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.management.MBeanHome;
import weblogic.management.WebLogicObjectName;
import weblogic.management.configuration.ApplicationMBean;
import weblogic.management.configuration.EJBComponentMBean;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.runtime.EJBComponentRuntimeMBean;
import weblogic.management.runtime.EJBPoolRuntimeMBean;
import weblogic.management.runtime.EJBRuntimeMBean;
import weblogic.management.runtime.EJBTransactionRuntimeMBean;
import weblogic.management.runtime.StatelessEJBRuntimeMBean;
import weblogic.jndi.Environment;

public final class EJBMonitor {

    private String url = "t3://localhost:7001";
    private String user = "weblogic";
    private String password = "weblogic";

    private MBeanHome   mBeanHome; // admin

    public EJBMonitor() throws Exception {
        Environment env = new Environment();
        env.setProviderUrl(url);
        env.setSecurityPrincipal(user);
        env.setSecurityCredentials(password);
        Context ctx = env.getInitialContext();

        mBeanHome = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
    }

    public void displayStatelessEJBPoolMissPercentages()
    throws Exception
    {
        String type = "StatelessEJBRuntime";
        Set beans = mBeanHome.getMBeansByType(type);
        System.out.println("Printing Stateless Session pool miss
```

```
                        percentages:");
        for(Iterator it=beans.iterator();it.hasNext();) {
            StatelessEJBRuntimeMBean rt = (StatelessEJBRuntimeMBean)it.next();
            displayEJBInfo(rt);
            EJBPoolRuntimeMBean pool = rt.getPoolRuntime();

            String missPercentage = "0";
            long missCount = pool.getMissTotalCount();
            if(missCount > 0) {
                missPercentage =
                        ""+(float)missCount/pool.getAccessTotalCount()*100;
            }
            System.out.println("Pool Miss Percentage: "+ missPercentage +"\n");
        }
    }

    public void displayStatefulEJBTransactionRollbackPercentages()
    throws Exception
    {
        String type = "StatefulEJBRuntime";
        Set beans = mBeanHome.getMBeansByType(type);
        System.out.println("Printing Stateful transaction rollback
                          percentages:");
        for(Iterator it=beans.iterator();it.hasNext();) {
            EJBRuntimeMBean rt = (EJBRuntimeMBean)it.next();
            displayEJBInfo(rt);
            EJBTransactionRuntimeMBean trans = rt.getTransactionRuntime();

             String rollbackPercentage = "0";
            long rollbackCount = trans.getTransactionsRolledBackTotalCount();
            if(rollbackCount > 0) {
                long totalTransactions = rollbackCount +
                        trans.getTransactionsCommittedTotalCount();
                rollbackPercentage =
                        ""+(float)rollbackCount/totalTransactions*100;
            }
            System.out.println("Transaction rollback percentage: "+
                        rollbackPercentage +"\n");
        }
    }

    private void displayEJBInfo(EJBRuntimeMBean rt) throws Exception {
        System.out.println("EJB Name: "+rt.getEJBName());
        EJBComponentRuntimeMBean compRTMBean =
                        (EJBComponentRuntimeMBean)rt.getParent();
        EJBComponentMBean compMBean = compRTMBean.getEJBComponent();
        ApplicationMBean appMBean = (ApplicationMBean)compMBean.getParent();
        System.out.println("Application Name: "+appMBean.getName());
        System.out.println("Component Name: "+compMBean.getName());
```

```
        WebLogicObjectName objName = rt.getObjectName();
        System.out.println("Server Name: "+objName.getLocation());
    }

    public static void main(String[] argv) throws Exception {
        EJBMonitor m = new EJBMonitor();
        m.displayStatelessEJBPoolMissPercentages();
        m.displayStatefulEJBTransactionRollbackPercentages();
    }
}
```

Viewing Runtime Information for Servlets

Instances of `ServletRuntimeMBean` provide access to information about how individual servlets are performing. For example, the `ServletRuntime.InvocationTotalCount` attribute indicates the number of times a servlet instance has been invoked.

Because the `WebLogicObjectName` for instances of `ServletRuntimeMBean` is dynamically generated each time a servlet type is instantiated, it is not feasible to register JMX listeners or monitors with each servlet. Instead, you can look up `ServletRuntime` MBeans and invoke `ServletRuntime` methods.

The general structure for `ServletRuntime` object names is as follows:
`domain``:Location=`*`dynamic-name`*`,ServerRuntime=`*`server`*`,Type=ServletRuntime`

The *`dynamic-name`* value includes the name of the server on which the servlet is deployed, the name of the application that contains the servlet, the class name of the servlet, and a number that is assigned to the specific servlet instance.

For example:
`medrec:Location=MedRecServer,Name=MedRecServer_MedRecServer_MainWAR_org.apache.struts.action.ActionServlet_549,ServerRuntime=MedRecServer,Type=ServletRuntime`

`ServletRuntime` MBeans are instantiated with an MBean hierarchy (see Figure 5-2):

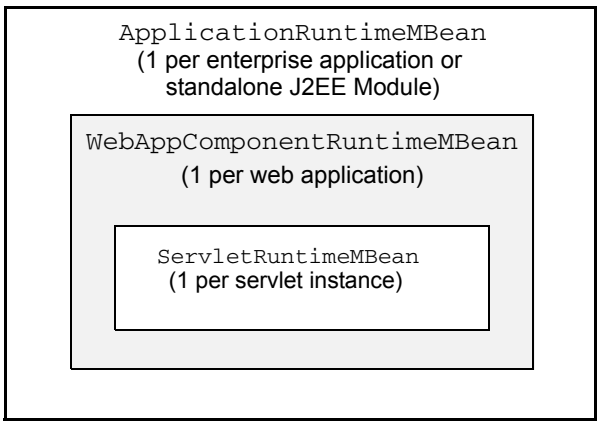
- Each enterprise application exposes its runtime data through an instance of `ApplicationRuntimeMBean`.

  If you deploy J2EE modules (such as EJBs or Web applications) without declaring them as components of an enterprise application, WebLogic Server creates a runtime wrapper enterprise application and deploys each module within its own wrapper application. The wrapper application name is the same as the module that it contains. For example, if you

deploy `myApp.WAR`, WebLogic Server wraps the web application in an enterprise application named `myApp`. You do not need to interact with this wrapper application; it is an artifact of the WebLogic Server deployment implementation.

- Each Web application exposes its runtime data through an instance of `WebAppComponentRuntimeMBean`.
- Each servlet instance exposes its runtime data through an instance of `ServletRuntimeMBean`.

Figure 5-2 Hierarchy of Application, Web Application, and Servlet Runtime MBeans



Although you can retrieve instances of `ServletRuntimeMBean` without walking the MBean hierarchy, BEA recommends that you use the hierarchical organization to retrieve only servlets within a specific Web application. (Depending on the number of servlets on a server instance, retrieving all `ServletRuntime` MBeans on a server instance can lead to poor performance.) After you retrieve servlets within a Web application, you can iterate through the list to retrieve monitoring data.

Example: Retrieving Runtime Information for Servlets

To retrieve the value of the `ServletRuntime.InvocationTotalCount` attribute for all instances of the action servlet within the sample `Patient` Web application (which is a component of the `MedRec` application), the sample class in Listing 5-9:

1. Connects to the Administration Server and retrieves the Administration `MBeanHome` interface.

The Administration `MBeanHome` interface provides access to all Web applications (and therefore all of their servlet instances) that are deployed in the domain.

If you want to retrieve runtime information only for the servlet instances on a specific server instance, you can connect to the specific server instance and retrieve the local `MBeanHome` interface. For more information, refer to "Example: Retrieving a Local MBeanHome from an Internal Client" on page 2-9.

2. To retrieve the `ApplicationRuntimeMBean` for the MedRec application, the class:

   a. Invokes `MBeanHome.getMBeansByType` to retrieve all `ApplicationRuntime` MBeans in the domain.

   b. For each `ApplicationRuntimeMBean`, it invokes `ApplicationRuntimeMBean.getApplicationName` and compares the returned value to the value of the `appName` variable.

   c. When it finds the `ApplicationRuntimeMBean` for the MedRec application, it returns the MBean.

   Note: Depending on the number of Web applications in your domain, this step might not be necessary. If your domain contains only a few Web applications, you can simply retrieve all `WebAppComponentRuntime` MBeans and iterate through this list to find a specific Web application.

3. To retrieve the `WebAppComponentRuntimeMBean` for the Patient Web application, the class:

   a. Invokes `ApplicationRuntimeMBean.lookupComponents` to retrieve all application components.

   b. Iterates through the list to retrieve only the Web application components (which are represented by `WebAppComponentRuntime` MBeans.

   c. For each `WebAppComponentRuntimeMBean`, it invokes `WebAppComponentMBean.getContextRoot` and compares the returned value with the value of the `ctxRoot` variable.

      The context root is a convenient, unique identifier for a Web application. If you deploy a Web application as part of an enterprise application, you specify the context root in the application's `application.xml` deployment descriptor. If you deploy a Web application as a standalone module, you define the context root in the Web application's `weblogic.xml` deployment descriptor.

For the Patient Web application, its context root is defined with the following XML elements from *WL_HOME*\samples\server\medrec\src\medrecEar\META-INF\application.xml:

```
<module>
  <web>
     <web-uri>patientWebApp</web-uri>
      <context-root>/patient</context-root>
  </web>
</module>
```

d. When it finds the `WebAppComponentRuntimeMBean` for the Patient Web application, it returns the MBean.

4. To find all instances of the action servlet within the Patient Web application, the code:

   a. Invokes `WebAppComponentMBean.getServlets` to retrieve all instances of `ServletRuntimeMBean`.

   b. For each `ServletRuntimeMBean`, it invokes `ServletRuntimeMBean.getServletName` and compares the returned value to the value of the `servletName` variable.

   c. When it finds a `ServletRuntimeMBean` that represents an instance of the action servlet, it returns the MBean.

5. To return the invocation count for each instance of the action servlet, the code invokes `ServletRuntimeMBean.getInvocationTotalCount` and prints the returned value to standard out.

Listing 5-9 Retrieving Invocation Count from Servlets in a Web Application

```
import java.util.Set;
import java.util.Iterator;
import java.util.regex.Pattern;
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServletRuntimeMBean;
import weblogic.management.runtime.ApplicationRuntimeMBean;
import weblogic.management.runtime.WebAppComponentRuntimeMBean;
import weblogic.management.runtime.ComponentRuntimeMBean;
```

```
public class ServletRuntime {
   public static void main(String[] args) {
      //url of the Administration Server
      String url = "t3://localhost:7001";
      String username = "weblogic";
      String password = "weblogic";
      String appName = "MedRecEAR";
      String ctxRoot = "/patient";
      String servletName = "action";

      try {
         MBeanHome home = getMBeanHome(url, username, password);
         ApplicationRuntimeMBean app = getApplicationRuntimeMBean
             (home, appName);
         WebAppComponentRuntimeMBean webapp =
             getWebAppComponentRuntimeMBean(app,ctxRoot);
         ServletRuntimeMBean servlet = getServletRuntimeMBean
             (webapp,servletName);
         System.out.println("Invocation count is " +
             servlet.getInvocationTotalCount());
      } catch (Exception e) {
         System.out.println("Exception caught: " + e);
      }
   }

   /**
   * Get an initial context and lookup the Admin MBean Home
   */
   private static MBeanHome getMBeanHome(String url,
      String username, String password) throws Exception
   {
      Environment env = new Environment();
      env.setProviderUrl(url);
      env.setSecurityPrincipal(username);
      env.setSecurityCredentials(password);
      Context ctx = env.getInitialContext();

      // Retrieve the Administration MBeanHome
      return (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
   }
```

```
/**
* Find the RuntimeMBean for an application
*/
private static ApplicationRuntimeMBean
   getApplicationRuntimeMBean(MBeanHome home, String appName)
   throws Exception
{
   //
   // Get the Set of RuntimeMBeans for all applications.
   //
   Set appMBeans = home.getMBeansByType("ApplicationRuntime");
   Iterator appIterator = appMBeans.iterator();
   //
   // Iterate over the Set and find the app you are interested in
   //
   while (appIterator.hasNext()) {
      ApplicationRuntimeMBean appRuntime = (ApplicationRuntimeMBean)
         appIterator.next();
      if (appName.equals(appRuntime.getApplicationName())) {
         return appRuntime;
      }
   }
   throw new Exception("Could not find RuntimeMBean for "+appName);
}

/**
* Find the RuntimeMBean for a web application within a given application
*/
private static WebAppComponentRuntimeMBean
   getWebAppComponentRuntimeMBean(ApplicationRuntimeMBean app,
   String ctxroot) throws Exception
{
   ComponentRuntimeMBean[] compMBeans = app.lookupComponents();
   if (compMBeans == null) {
      throw new Exception("Application has no components");
   }
   for (int i=0; i<compMBeans.length; i++) {
      if (compMBeans[i] instanceof WebAppComponentRuntimeMBean) {
```

```
            WebAppComponentRuntimeMBean webMBean =
              (WebAppComponentRuntimeMBean)compMBeans[i];
            if (ctxroot.equals(webMBean.getContextRoot())) {
               return webMBean;
            }
         }
      }
      throw new Exception("Could not find web application with context
         root "+ctxroot);
   }

   /**
   * Find the RuntimeMBean for a servlet within a given web application
   */
   private static ServletRuntimeMBean
      getServletRuntimeMBean(WebAppComponentRuntimeMBean webapp,
      String servletName) throws Exception
   {
      ServletRuntimeMBean[] svltMBeans = webapp.getServlets();
      if (svltMBeans == null) {
         throw new Exception("No servlets in "+webapp.getComponentName());
      }
      for (int j=0; j<svltMBeans.length; j++) {
         if (servletName.equals(svltMBeans[j].getServletName())) {
            return svltMBeans[j];
         }
      }
      throw new Exception("Could not find servlet named "+servletName);
   }
}
```

CHAPTER 6

Using WebLogic Server MBean Notifications and Monitors

To report changes in configuration and runtime information, all WebLogic Server MBeans emit JMX notifications. A **notification** is a JMX object that describes a state change or some other specific condition that has occurred in an underlying resource.

You can create Java classes called **listeners** that listen for these notifications. For example, your application can include a listener that receives notifications when applications are deployed, undeployed, or redeployed.

The following sections describe working with notifications and listeners:

How Notifications are Broadcast and Received

All WebLogic Server MBeans implement the `javax.management.NotificationBroadcaster` interface, which enable them to emit different types of notification objects depending on the type of event that occurs. For example, MBeans emit notifications when the values of their attributes change.

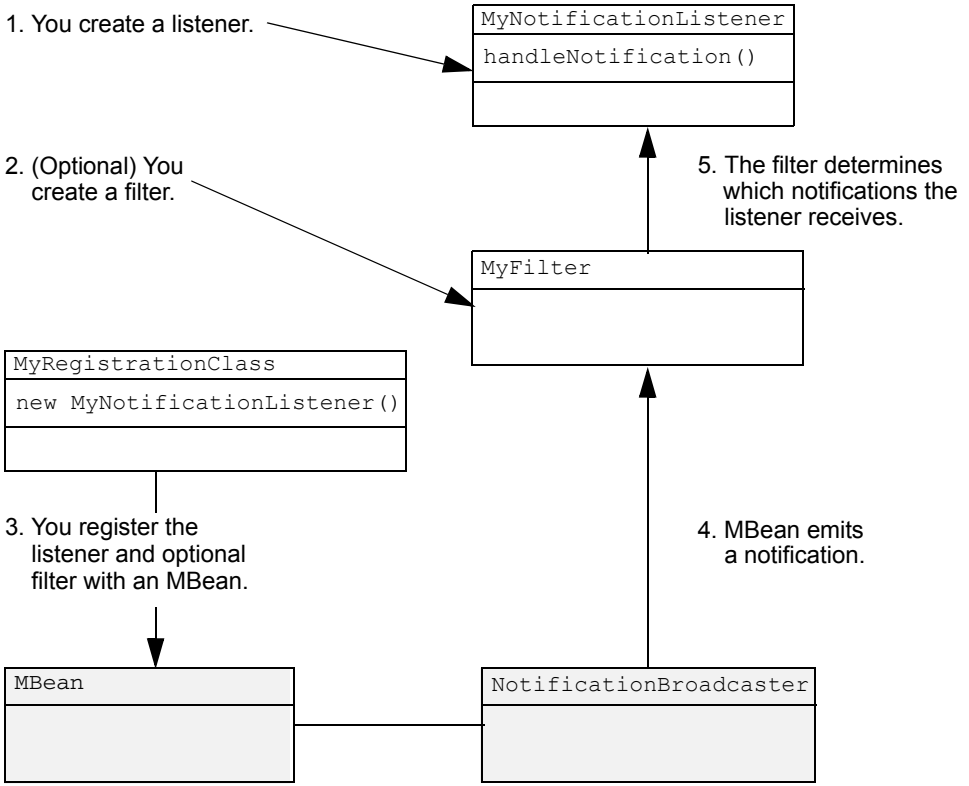
To listen for these notifications, you create a listener class that implements `javax.management.NotificationListener`.

By default, your listener receives all notifications that the MBean emits. However, typically, you want your listener to retrieve only specific notifications. For example, the `LogBroadCasterRuntime` MBean emits a notification each time a WebLogic Server instance generates a log message. Usually you listen for only specific log messages, such as messages of specific severity level. To limit the notifications that your listener receives, you can create a notification filter.

After creating your listener and optional filter, you register the classes with the MBeans from which you want to receive notifications.

Figure 6-1 shows a basic system in which a `NotificationListener` receives only a subset of the notifications that an MBean broadcasts.

Figure 6-1 Receiving Notifications from an MBean



For a complete explanation of JMX notifications and how they work, download the JMX 1.0 specification from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html.

Monitoring Changes in MBeans

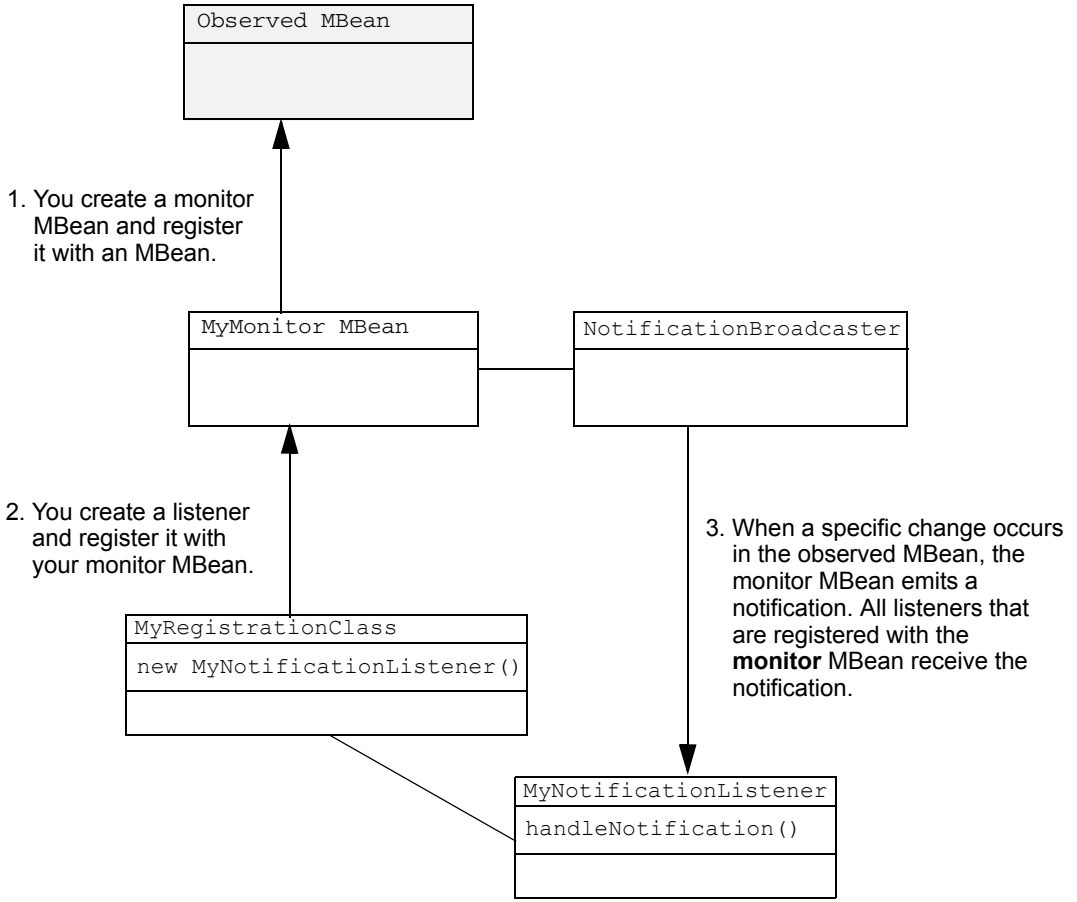
WebLogic Server includes a set of **monitor MBeans** that can be configured to periodically observe MBeans and emit JMX notifications only if a specific MBean attribute has changed beyond a specific threshold. A monitor MBean can observe the exact value of an attribute in an MBean, or optionally, the difference between two consecutive values of a numeric attribute. The value that a monitor MBean observes is called the **derived gauge**.

When the value of the derived gauge satisfies a set of conditions, the monitor MBean emits a specific notification type. Monitors can also send notifications when certain error cases are encountered while monitoring an attribute value.

To use monitor MBeans, you configure and register a monitor with a WebLogic Server MBean. Then you create a listener class and register the class with the **monitor MBean**. Because monitor MBeans emit only very specific types of notification, you usually do not use filters when listening for notifications from monitor MBeans.

Figure 6-2 shows a basic system in which a monitor MBean is registered with a WebLogic Server MBean. A `NotificationListener` is registered with the monitor MBean, and it receives notifications when the conditions within the monitor MBean are satisfied.

Figure 6-2 Monitor MBeans



Best Practices: Listening Directly Compared to Monitoring

WebLogic Server provides two ways to be notified about changes in an MBean: you can create a listener and register it directly with an MBean (see Figure 6-1), or you can configure a monitor MBean that periodically observes an MBean and sends notifications when an attribute value satisfies criteria that you specify (see Figure 6-2). The method that you choose depends mostly on the complexity of the situations in which you want to receive notifications.

If your requirements are simple, registering a listener directly with an MBean is the preferred technique. The `NotificationListener` and `NotificationFilter` interfaces, which are

classes that you implement in your listener and filter, provide few facilities for comparing values with thresholds and other values. You must create you own code to evaluate the data within notifications and respond accordingly. However, the advantage of registering a listener directly with an MBean is that the MBean pushes its notifications to your listener and you are notified of a change almost immediately.

If your notification requirements are sufficiently complex, or if you want to monitor some set of changes that are not directly associated with a single change in the value of an MBean attribute, use a monitor MBean. The monitor MBeans provide a rich set of tools for comparing data and sending notifications only under highly specific circumstances. However, the monitor periodically polls the observed MBean for changes in attribute value and you are notified of a change only as frequently as the polling interval that you specify.

Best Practices: Commonly Monitored Attributes

The attributes in Table 6-1 provide a general overview of the performance of WebLogic Server. You can monitor these attributes either by creating a listener and registering it directly with the MBeans that contain the attributes or by configuring monitor MBeans.

To create and register a listener or to configure monitor MBeans, you must provide the `WebLogicObjectName` of the MBean that contains the attributes you want to monitor. (See "Registering a Notification Listener and Filter" on page 6-15 and "Instantiating the Monitor and Listener" on page 6-26.)

Use the information in Table 6-1 to construct the `WebLogicObjectName` for each MBean. In the table, *`domain`* refers to the name of the WebLogic Server domain, and *`server`* refers to the name of the WebLogic Server instance that hosts the MBean you want to monitor.

Table 6-1 Commonly Monitored WebLogic Server Attributes

MBean and Attribute Names	Description
MBean Type: `ServerRuntime` Attribute Name: `State` `WebLogicObjectName` for the MBean: *`domain`*`:Location=`*`server`*`,Name=`*`server`*`,Type=ServerRuntime` For example: `medrec:Location=MedRecServer,Name=MedRecServer,Type=ServerRuntime`	Indicates whether the server is in an `Initializing`, `Suspended`, `Running`, or `ShuttingDown` state.

Table 6-1 Commonly Monitored WebLogic Server Attributes

MBean and Attribute Names	Description
MBean Type: `ServerRuntime` Attribute Name: `OpenSocketsCurrentCount` `WebLogicObjectName` for the MBean: See the previous row in this table.	Use these two attributes together to compare the current activity on the server's listen ports to the total number of requests that can be backlogged on the ports. Note that the attributes are located in two separate MBeans: ■ `OpenSocketsCurrentCount` is in the `ServerRuntime` MBean. ■ `AcceptBacklog` is in the `Server` configuration MBean.
MBean Type: `Server` Attribute Name: `AcceptBacklog` `WebLogicObjectName` for the MBean: *`domain`*`:Name=`*`server`*`,Type=Server` For example: `medrec:Name=MedRecServer,Type=Server`	

MBean Type: `ExecuteQueueRuntime`
Attribute Name:
`ExecuteThreadCurrentIdleCount`
`WebLogicObjectName` for the MBean:
`domain``:Location=`*`server`*`,`
`Name=weblogic.kernel.Default,`
`ServerRuntime=`*`server`*`,`
`Type=ExecuteQueueRuntime`
For example:
`medrec:Location=MedRecServer,Name=`
`weblogic.kernel.Default,`
`ServerRuntime=MedRecServer,`
`Type=ExecuteQueueRuntime`
	Displays the number of threads in a server's default execute queue that are taking up memory space but are not being used to process data.
	You can create multiple execute queues on a server instance to optimize the performance of critical applications, but the `default` execute queue is available by default. For more information, refer to "Using Execute Queues to Control Thread Usage."

MBean Type:`ExecuteQueueRuntime` Attribute Name: `PendingRequestCurrentCount` `WebLogicObjectName` for the MBean: See the previous row in this table.	Displays the number of requests waiting in a server's default execute queue.

Table 6-1 Commonly Monitored WebLogic Server Attributes

MBean and Attribute Names	Description
MBean Type: `JVMRuntime` Attribute Name: `HeapSizeCurrent` `WebLogicObjectName` for the MBean: *`domain`*`:Location=`*`server`*`,Name=`*`server`*`,ServerRuntime=`*`server`*`,Type=JVMRuntime` For example: `medrec:Location=MedRecServer,Name=MedRecServer,ServerRuntime=MedRecServer,Type=JVMRuntime`	Displays the amount of memory (in bytes) that is currently available in the server's JVM heap. For more information, refer to "Tuning Java Virtual Machines (JVMs)."
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `ActiveConnectionsCurrentCount` `WebLogicObjectName` for the MBean: *`domain`*`:Location=`*`server`*`,Name=`*`poolName`*`,ServerRuntime=`*`server`*`,Type=JDBCConnectionPoolRuntime` where *`poolName`* is the name that you gave to the connection pool when you created it. For example: `medrec:Location=MedRecServer,Name=MedRecPool-PointBase,ServerRuntime=MedRecServer,Type=JDBCConnectionPoolRuntime`	Displays the current number of active connections in a JDBC connection pool. For more information, refer to "How JDBC Connection Pools Enhance Performance."
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `ActiveConnectionsHighCount` `WebLogicObjectName` for the MBean: See the previous row in this table.	The high water mark of active connections in a JDBC connection pool. The count starts at zero each time the connection pool is instantiated.
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `LeakedConnectionCount`	Notify a listener when the total number of leaked connections reaches a predefined threshold. Leaked connections are connections that have been checked out but never returned to the connection pool via a `close()` call; it is important to monitor the total number of leaked connections, as a leaked connection cannot be used to fulfill later connection requests.

Table 6-1 Commonly Monitored WebLogic Server Attributes

MBean and Attribute Names	Description
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `ActiveConnectionsCurrentCount`	Notify a listener when the current number of active connections to a specified JDBC connection pool reaches a predefined threshold.
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `ConnectionDelayTime`	Notify a listener when the average time to connect to a connection pool exceeds a predefined threshold.
MBean Type: `JDBCConnectionPoolRuntime` Attribute Name: `FailuresToReconnect`	Notify a listener when the connection pool fails to reconnect to its datastore. Applications may notify a listener when this attribute increments, or when the attribute reaches a threshold, depending on the level of acceptable downtime.

Listening for Notifications from WebLogic Server MBeans: Main Steps

To listen for the notifications that WebLogic Server MBeans emit directly:

1. Determine which notification type you want to listen for. See "WebLogic Server Notification Types" on page 6-9.
2. Create a listener class in your application. See "Creating a Notification Listener" on page 6-10.
3. Optionally create a filter class, which specifies the types of notifications that the listener receives from the MBeans. See "Creating a Notification Filter" on page 6-13.
4. Create an additional class that registers your listener and filter with the MBeans whose notifications you want to receive. See "Registering a Notification Listener and Filter" on page 6-15.

WebLogic Server Notification Types

WebLogic Server MBeans implement the `javax.management.NotificationBroadcaster` interface, which enable them to emit different types of notification objects depending on the type of event that occurs:

- When an MBean's attribute value changes, it emits a `javax.management.AttributeChangeNotification` object.
- When a WebLogic Server resource generates a log message, the server's `LogBroadcasterRuntimeMBean` emits a notification of type `weblogic.management.WebLogicLogNotification`. For more information about `WebLogicLogNotification`, refer to the WebLogic Server Javadoc.
- When MBeans are registered or unregistered, the WebLogic Server JMX services emit notifications of type `javax.management.MBeanServerNotification`.
- If an MBean attribute is an array, when you invoke the MBean's `add`*`AttributeName`* method to add an element to the array, the MBean emits a `weblogic.management.AttributeAddNotification` object. One example of an MBean that exposes `add`*`AttributeName`* methods is `weblogic.management.configuration.XMLRegistryMBean`. For more information, refer to the WebLogic Server Javadoc.
- If an MBean attribute is an array, when you invoke the MBean's `remove`*`AttributeName`* method to remove an element from the array, the MBean emits a `weblogic.management.AttributeRemoveNotification` object.

For more information about the `javax.management` notification types, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

For more information about the `weblogic.management` notification types, refer to the Javadoc for AttributeAddNotification and AttributeRemoveNotification.

Creating a Notification Listener

To create a notification listener:

1. Create a class that implements **one** of the following:
   - For a client that runs within the same JVM as WebLogic Server, implement `javax.management.NotificationListener`.
   - For a client that runs in a remote JVM, implement `weblogic.management.RemoteNotificationListener`.

     `RemoteNotificationListener` extends `javax.management.NotificationListener` and `java.rmi.Remote`, making MBean notifications available to external clients via RMI.

2. Within the class, add **one** of the following:
   - For a client that runs within the same JVM as WebLogic Server, add a `NotificationListener.handleNotification(Notification notification, java.lang.Object handback)` method.
   - For a client that runs within the same JVM as WebLogic Server, add a `RemoteNotificationListener.handleNotification(Notification notification, java.lang.Object handback)` method.

   Note: Your implementation of this method should return as soon as possible to avoid blocking its notification broadcaster.

3. To retrieve data from the notification objects that the listener receives, within your `handleNotification` method, invoke `javax.management.Notification` methods on the notification objects.

   For example, to retrieve the time stamp associated with the notification, invoke `notification.getTimeStamp()`.

   Because all notification types extend `javax.management.Notification`, the following `Notification` methods are available for all notifications:

   - `getMessage()`
   - `getSequenceNumber()`
   - `getTimeStamp()`
   - `getType()`
   - `getUserData()`

   For more information on `Notification` methods, refer to the `javax.management.Notification` Javadoc in the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

4. Most notification types provide additional methods for retrieving data that is specific to the notification. For example, `WebLogicLogNotification` provides methods for retrieving specific attributes of WebLogic Server log messages, such as `getSeverity()`, which retrieves the severity level that the log message specifies.

   If you want to retrieve data that is specific to a notification type (and therefore not retrievable through the standard `javax.management.Notification` methods):

   a. Add logic within the `handleNotification` method to filter through the notifications and select only notifications of a specific type.

b. Invoke methods that the notification type provides to extract data from the notification object.

   For example:

```
if(notification instanceof MonitorNotification) {
  MonitorNotification monitorNotification = (MonitorNotification)
                                                    notification;
  System.out.println("This notification is a MonitorNotification");
  System.out.println("Observed Attribute: " +
                       monitorNotification.getObservedAttribute() );
}
```

In addition to the previous steps, consider the following while creating your `NotificationListener` class:

- Unless you create and use a notification filter, your listener receives all notifications (of all notification types) from the MBeans with which it is registered.

  Instead of using one listener for all possible notifications that an MBean emits, the best practice is to use a combination of filters and listeners. While having multiple listeners adds to the amount of time for initializing the JVM, the trade-off is ease of code maintenance.

- If your WebLogic Server environment contains multiple instances of MBean types that you want to monitor, you can create one notification listener and then create as many registration classes as MBean instances that you want to monitor.

  For example, if your WebLogic Server domain contains three JDBC connection pools, you can create one listener class that listens for `AttributeChangeNotifications`. Then, you create three registration classes. Each registration class registers the listener with a specific instance of a `JDBCConnectionPoolRuntime` MBean.

- While the `handleNotification` method signature includes an argument for a handback object, your listener does not need to retrieve data from or otherwise manipulate the handback object. It is an opaque object that helps the listener to associate information regarding the MBean emitter.

The following example creates a remote listener. Then the listener receives a `AttributeChangeNotification` object, it uses `AttributeChangeNotification` methods to retrieve the name of the attribute with a changed value, and the old and new values.

Listing 6-1 Notification Listener

```
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;
import weblogic.management.RemoteNotificationListener;
import javax.management.AttributeChangeNotification;

public class MyListener implements RemoteNotificationListener {

    public  void handleNotification(Notification notification, Object obj) {

        if(notification instanceof AttributeChangeNotification) {
            AttributeChangeNotification attributeChange =
                     (AttributeChangeNotification) notification;
            System.out.println("This notification is an
                      AttributeChangeNotification");
            System.out.println("Observed Attribute: " +
                                       attributeChange.getAttributeName() );
            System.out.println("Old Value: " + attributeChange.getOldValue() );
            System.out.println("New Value: " + attributeChange.getNewValue() );
        }
    }
}
```

Creating a Notification Filter

To create and register a filter:

1. Create a serializable class that implements `javax.management.NotificationFilter`.

   Optionally import the `javax.management.NotificationFilterSupport` class, which provides utility methods for filtering notifications. For more information about using these methods, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

   The filter needs to be serializable only if it is used in a remote notification listener. A class that is used with RMI must be serializable so it can be deconstructed and reconstructed in remote JVMs.

2. Use the `isNotificationEnabled(Notification notification)` method to indicate whether the serializable object returns a true value when a set of conditions are satisfied.

If the boolean returns true, then the filter forwards the notification to the listener with which the filter is registered.

3. (Optional) You can include code that retrieves data from notifications and carries out actions based on the data in the notification. For example, your filter can use `javax.management.AttributeChangeNotification` methods to view the new value of a specific attribute. If the value is over a threshold that you specify, you can use JavaMail API to send e-mail to an administrator.

Listing 6-2 provides an example `NotificationFilter` that forwards only notifications of type `AttributeChangeNotification`.

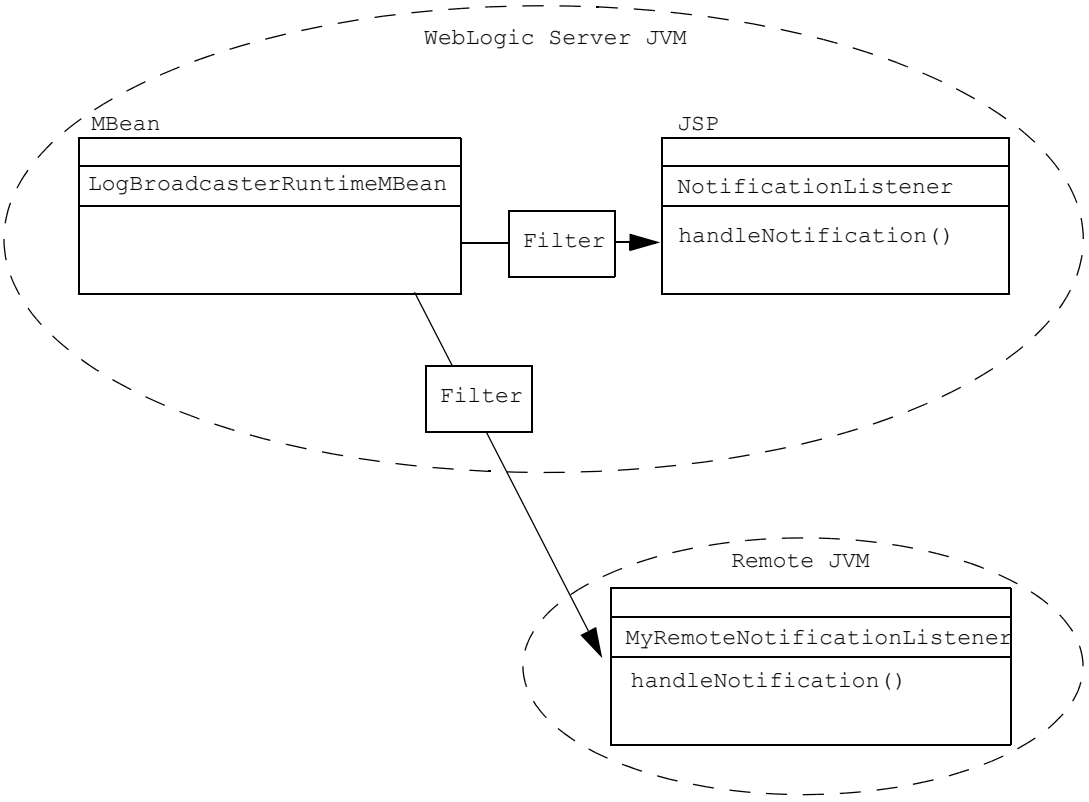
Listing 6-2 Example Notification Filter

```
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.AttributeChangeNotification;

public class MyHiCountFilter implements NotificationFilter,
                 java.io.Serializable {

    public boolean isNotificationEnabled(Notification notification) {
        if (!(notification instanceof AttributeChangeNotification)) {
            return false;
        }
        AttributeChangeNotification acn =
                    (AttributeChangeNotification)notification;
        acn.getAttributeName().equals("ActiveConnectionsHighCount"); {
            return true;
        }
    }
}
```

Adding Filter Classes to the Server Classpath

If you create a filter for a listener that runs in a remote JVM, you can add the filter's classes to the classpath of the server instance from which you are listening for notifications. Although the listener runs in the remote JVM, adding the filter's classes to the server' s classpath minimizes the transportation of serialized data between the filter and the listener. (See Figure 6-3.)

Figure 6-3 Filters Can Run on WebLogic Server



Registering a Notification Listener and Filter

After you implement a notification listener class and optional filter class, you create an additional class that registers your listener and filter with an MBean instance. You must create one registration class for each MBean instance that you want to monitor.

To register a notification listener and filter:

1. Create a class that retrieves the `MBeanHome` interface and then uses `MBeanHome` to retrieve the `MBeanServer` interface.

   If you want to register a listener and filter with an Administration MBean, you must retrieve the Administration `MBeanHome`, which resides only on the Administration Server.

If you want to register with a Local Configuration MBean or a Runtime MBean, you must retrieve the Local `MBeanHome` for the server instance that hosts the MBean.

2. Instantiates the listener class and filter class that you created.
3. Constructs the `WebLogicObjectName` of the MBean with which you want to register.

   For a list of commonly monitored MBeans and their `WebLogicObjectName`, refer to Table 6-1, "Commonly Monitored WebLogic Server Attributes," on page 6-6.

4. Registers the listener and filter by passing the `WebLogicObjectName`, listener class, and filter class to the `addNotificationListener()` method of the `MBeanServer` interface.

   While Figure 6-1 illustrates registering a listener and filter directly with an MBean (which you can do by calling the MBean's `addNotificationListener()` method), in practice it is preferable to use the `addNotificationListener()` method of the `MBeanServer` interface, which saves the trouble of looking up a particular MBean simply for registration purposes.

The following example is a registration class that runs in a remote JVM. If the class ran within the same JVM as a WebLogic Server instance, the code for retrieving the `MBeanHome` interface would be simpler. For more information, refer to "Accessing an MBeanHome Interface" on page 2-4.

The example class registers the listener from Listing 6-1 and filter from Listing 6-2 with the `Server` Administration MBean for a server instance named Server1. In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

The example class also includes some code the keep the class active until it receives a notification. Usually this code is not necessary because a listener class runs in the context of some larger application that is responsible for invoking the class and keeping it active. It is included here so you can easily compile and see the example working.

Listing 6-3 Registering a Listener for an Administration MBean

```
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.management.ObjectName;
import javax.management.Notification;
```

```
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicMBean;
import weblogic.management.WebLogicObjectName;
import weblogic.management.RemoteMBeanServer;
import weblogic.management.configuration.ServerMBean;

public class listener {

    public static void main(String[] args) {

        MBeanHome home = null;
        RemoteMBeanServer rmbs = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "Server1";
        String username = "weblogic";
        String password = "weblogic";

        //Using MBeanHome to get MBeanServer.
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //Getting the Administration MBeanHome.
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Admin MBeanHome: " + home );
            rmbs = home.getMBeanServer();
        } catch (Exception e) {
            System.out.println("Caught exception: " + e);
        }

        try {
            //Instantiating your listener class.
            MyListener listener = new MyListener();
            MyFilter filter = new MyFilter();

            //Constructing the WebLogicObjectName of the MBean that you want
            //to listen to.
            WebLogicObjectName mbeanName = new WebLogicObjectName(serverName,
                              "Server",home.getDomainName());
            System.out.println("Created WebLogicObjectName: " + mbeanName);

            //Passing the name of the MBean and your listener class to the
            //addNotificationListener method of MBeanServer.
            rmbs.addNotificationListener(mbeanName, listener, filter, null);
            System.out.println("\n[myListener]: Listener registered ...");
```

```
            //Keeping the remote client active.
            System.out.println("pausing..........");
            System.in.read();
        } catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

Listening for Configuration Auditing Messages: Main Steps

You can configure the Administration Server to emit a log message when a user changes the configuration or invokes management operations on any resource within a domain. For example, if a user disables SSL on a Managed Server in a domain, the Administration Server emits a log message. These messages provide an audit trail of changes within a domain's configuration (configuration auditing). See "Configuration Auditing" in *Administration Console Online Help*.

To create and use a JMX listener and filter that respond to configuration auditing messages:

1. Create and compile a notification listener that extracts information from WebLogic Server log messages.

   See "Notification Listener for Configuration Auditing Messages" on page 6-19.

2. Create and compile a notification filter that selects only configuration auditing messages.

   See "Notification Filter for Configuration Auditing Messages" on page 6-19.

3. Create and compile a class that registers the listener and filter with the Administration Server's `LogBroadcasterRuntime` MBean. This is the MBean that a WebLogic Server instance uses to broadcast its log messages as JMX notifications.

   See "Registration Class for Configuration Auditing Messages" on page 6-20.

4. Add the notification filter to the classpath for the Administration Server.

   If the notification listener runs within the Administration Server's JVM (for example, if it runs as a startup class), add the notification listener and registration class to the Administration Server's classpath as well.

5. Invoke the registration class or configure it as a startup class for the Administration Server.

   See "Startup and Shutdown Classes" in *Administration Console Online Help*.

Notification Listener for Configuration Auditing Messages

Like the notification listener in Listing 6-1, the listener in Listing 6-4 implements `RemoteNotificationListener` and its `handleNotification` method.

Because all configuration auditing messages are of type `WebLogicLogNotification`, the listener in Listing 6-4 imports the `WebLogicLogNotification` interface and uses its methods to retrieve information within each configuration auditing message.

Listing 6-4 Notification Listener for Configuration Auditing Messages

```
import javax.management.Notification;
import javax.management.NotificationListener;
import weblogic.management.RemoteNotificationListener;
import weblogic.management.logging.WebLogicLogNotification;

public class ConfigAuditListener implements RemoteNotificationListener {
    public  void handleNotification(Notification notification, Object obj) {
            WebLogicLogNotification changeNotification =
                    (WebLogicLogNotification) notification;
           System.out.println("A user has attempted to change the configuration
                    of a WebLogic Server domain.");
            System.out.println("Admin Server Name: " +
                    changeNotification.getServername() );
            System.out.println("Time of attempted change:" +
                    changeNotification.getTimeStamp() );
            System.out.println("Message details:" +
                    changeNotification.getMessage() );
            System.out.println("Message ID string:" +
                    changeNotification.getMessageId() );
    }
}
```

Notification Filter for Configuration Auditing Messages

Without a notification filter, the listener in Listing 6-4 would print the Server Name, Timestamp, and Message Text for all messages that the Administration Server broadcast.

To forward only the configuration auditing message that indicates a resource has been modified, the filter in Listing 6-5 uses the `WebLogicLogNotification.getMessageId` method to retrieve the message ID of all incoming log notifications.

The resource-change configuration auditing message is identified by the message ID `159904` (see Configuration Auditing" in *Administration Console Online Help*). If the message ID value in an incoming log notification matches the configuration auditing message ID, the filter evaluates as `true` and forwards the message to its registered listener.

Listing 6-5 Notification Filter for Configuration Auditing Messages

```
import javax.management.Notification;
import javax.management.NotificationFilter;
import weblogic.management.logging.WebLogicLogNotification;

public class ConfigAuditFilter implements NotificationFilter ,
                 java.io.Serializable{
    int configChangedId = 159904;

    public boolean isNotificationEnabled(Notification notification) {
        if (!(notification instanceof WebLogicLogNotification)) {
            return false;

        }

        WebLogicLogNotification wln =
                    (WebLogicLogNotification)notification;
        int messageId = wln.getMessageId();
        if (configChangedId == messageId) {
                return true;

        } else {
          return false;
        }
    }
}
```

Registration Class for Configuration Auditing Messages

The class in Listing 6-6 registers the notification listener and filter with the `LogBroadcasterRuntime` MBean of the Administration Server. This MBean is a singleton in each instance of WebLogic Server and is always named `TheLogBroadcaster`.

Listing 6-6 Registration Class for Configuration Auditing Messages

```
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.management.ObjectName;
import javax.management.Notification;
import weblogic.jndi.Environment;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicMBean;
import weblogic.management.WebLogicObjectName;
import weblogic.management.RemoteMBeanServer;
import weblogic.management.configuration.ServerMBean;

public class ListenRegistration {
    public static void main(String[] args) {
        MBeanHome home = null;
        RemoteMBeanServer rmbs = null;

        //domain variables
        String url = "t3://localhost:7001";
        String serverName = "examplesServer";
        String username = "weblogic";
        String password = "weblogic";

        //Using MBeanHome to get MBeanServer.
        try {
            Environment env = new Environment();
            env.setProviderUrl(url);
            env.setSecurityPrincipal(username);
            env.setSecurityCredentials(password);
            Context ctx = env.getInitialContext();

            //Getting the Administration MBeanHome.
            home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            System.out.println("Got the Admin MBeanHome: " + home );
            rmbs = home.getMBeanServer();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e);
        }

        try {
            //Instantiating your listener class.
            ConfigAuditListener listener = new ConfigAuditListener();
            ConfigAuditFilter filter = new ConfigAuditFilter();
```

```
            //Constructing the WebLogicObjectName of the MBean that you want
            //to listen to.

            WebLogicObjectName mbeanName = new WebLogicObjectName(
                      "TheLogBroadcaster",
                      "LogBroadcasterRuntime",
                      home.getDomainName(),
                      serverName );
            System.out.println("Created WebLogicObjectName: " + mbeanName);

            //Passing the name of the MBean and your listener class to the
            //addNotificationListener method of MBeanServer.
            rmbs.addNotificationListener(mbeanName, listener, filter, null);
            System.out.println("\n[myListener]: Listener registered ...");

            //Keeping the remote client active.
            System.out.println("pausing..........");
            System.in.read();
        } catch(Exception e) {
            System.out.println("Exception: " + e);

        }
    }
}
```

Using Monitor MBeans to Observe Changes: Main Steps

To configure and use monitor MBeans:

1. Choose a monitor MBean type that matches the type of data you want to observe. "Choosing a Monitor MBean Type" on page 6-22
2. Create a listener class that can listen for notifications from monitor MBeans. See "Creating a Notification Listener for a Monitor MBean" on page 6-25.
3. Create a class that configures a monitor MBean, registers your listener class with the monitor MBean, and then registers the monitor MBean with an observed MBean. "Instantiating the Monitor and Listener" on page 6-26

Choosing a Monitor MBean Type

WebLogic Server provides monitor MBeans that are specialized to observe changes in specific data types. You must configure and instantiate the type of monitor MBean that matches the type of the object that an MBean returns for an attribute value. For example, a monitor MBean based

on the `StringMonitor` type can observe an attribute that is declared as an `Object` as long as actual values of the attributes are `String` instances, as determined by the `instanceof` operator.

To choose a monitor type:

1. Determine the type of object that is returned by the MBean attribute that you want to observe by doing any of the following:
   - Refer to the WebLogic Server Javadoc.
   - Use the `weblogic.Admin GET` command, which provides information about the MBean that you specify. For more information, refer to "MBean Management Command Reference" in *Configuring and Managing WebLogic Server*.
   - Use the `javap` command on the MBean you are monitoring. The `javap` command is a standard Java utility that disassembles a class file.
2. Choose a monitor type from the following table.

Table 6-2 Monitor MBeans and Observed Object Types

A Monitor MBean of This Type	Observes This Object Type
`CounterMonitor`	`Integer`
`GaugeMonitor`	Integer or floating-point (`Byte`, `Integer`, `Short`, `Long`, `Float`, `Double`)
`StringMonitor`	`String`

For more information about monitor types, refer to the JMX 1.0 specification, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

Monitor Notification Types

Each type of monitor MBean emits specific types of `javax.management.monitor.MonitorNotification` notifications. For any given notification, you can use the `MonitorNotification.getType()` method to determine its type.

The following table describes the type of notifications that monitor MBeans emit.

Table 6-3 Monitor MBeans and MonitorNotification Types

A Monitor MBean of This Type	Emits This MonitorNotification Type
`CounterMonitor`	A counter monitor emits a `jmx.monitor.counter.threshold` when the value of the counter reaches or exceeds a threshold known as the comparison level.
`GaugeMonitor`	• If the observed attribute value is **increasing** and becomes equal to or greater than the high threshold value, the monitor emits a notification type of `jmx.monitor.gauge.high`. Subsequent crossings of the high threshold value do not cause further notifications unless the attribute value becomes equal to or less than the low threshold value. • If the observed attribute value is **decreasing** and becomes equal to or less than the low threshold value, the monitor emits a notification type of `jmx.monitor.gauge.low`. Subsequent crossings of the low threshold value do not cause further notifications unless the attribute value becomes equal to or greater than the high threshold value.
`StringMonitor`	• If the observed attribute value **matches** the string to compare value, the monitor emits a notification type of `jmx.monitor.string.matches`. Subsequent matches of the string to compare values do not cause further notifications unless the attribute value differs from the string to compare value. • If the attribute value **differs** from the string to compare value, the monitor emits a notification type of `jmx.monitor.string.differs`. Subsequent differences from the string to compare value do not cause further notifications unless the attribute value matches the string to compare value.

Error Notification Types

All monitors can emit the following notification types to indicate error cases:

- `jmx.monitor.error.mbean`, which indicates that the observed MBean is not registered in the MBean Server. The observed object name is provided in the notification.
- `jmx.monitor.error.attribute`, which indicates that the observed attribute does not exist in the observed object. The observed object name and observed attribute name are provided in the notification.
- `jmx.monitor.error.type`, which indicates that the object instance of the observed attribute value is `null` or not of the appropriate type for the given monitor. The observed object name and observed attribute name are provided in the notification.

- `jmx.monitor.error.runtime`, which contains exceptions that are thrown while trying to get the value of the observed attribute (for reasons other than the cases described above).

The counter and the gauge monitors can also emit the following `jmx.monitor.error.threshold` notification type under the following circumstances:

- For a counter monitor, when the threshold, the offset, or the modulus is not of the same type as the observed counter attribute.
- For a gauge monitor, when the low threshold or high threshold is not of the same type as the observed gauge attribute.

Creating a Notification Listener for a Monitor MBean

As any other MBean, monitor MBeans emit notifications by implementing `javax.management.NotificationBroadcaster`. To create a listener for notifications from a monitor MBean, create a class that does the following:

1. Implements `NotificationBroadcaster` or `weblogic.management.RemoteNotificationListener`.
2. Includes the `NotificationListener.handleNotification()` or the `RemoteNotificationListener.handleNotification()` method.

You can register the same notification listener with instances of `LogBroadcasterMBean`, monitor MBeans, or any other MBean.

The example below creates a listener object for an application that runs in a JVM outside the WebLogic Server JVM. It includes logic that outputs additional messages when it receives notifications from monitor MBeans. You could further refine the logic so that listener responds differently to the different types of monitor notifications described in "Monitor Notification Types" on page 6-23.

Listing 6-7 Listener for Monitor Notifications

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.monitor.MonitorNotification;
import weblogic.management.RemoteNotificationListener;

public class CounterListener implements RemoteNotificationListener {
   public void handleNotification(Notification notification ,Object obj) {
      System.out.println("\n\n Notification Received ...");
```

```
      System.out.println("Type=" + notification.getType() );
      System.out.println("Message=" + notification.getMessage() );
      System.out.println("SequenceNumber=" +
         notification.getSequenceNumber());
      System.out.println("Source=" + notification.getSource());
      System.out.println("Timestamp=" + notification.getTimeStamp() + "\n" );
      if(notification instanceof MonitorNotification) {
         MonitorNotification monitorNotification =
            MonitorNotification)notification;
         System.out.println("This notification is a MonitorNotification");
         System.out.println("Observed Attribute: " +
            monitorNotification.getObservedAttribute() );
         System.out.println("Observed Object: " +
            monitorNotification.getObservedObject() );
         System.out.println("Trigger value: " +
            monitorNotification.getTrigger() );
      }
   }
}
```

Instantiating the Monitor and Listener

The steps you take to register a monitor MBean with an observed MBean differ depending on whether you are registering the monitor MBean on a single server instance or on multiple server instances in a domain.

Note: Because WebLogic Server does not provide type-safe stubs for monitor MBeans, you must use standard JMX design patterns in which your JMX client uses the `MBeanServer` interface to get and set attributes and invoke operations on the monitor MBean.

The following sections provide examples for both tasks:

- "Example: Monitoring an MBean on a Single Server" on page 6-26
- "Example: Monitoring Instances of an MBean on Multiple Servers" on page 6-30

Example: Monitoring an MBean on a Single Server

The following example creates a counter monitor for the `ServicedRequestTotalCount` attribute of the `ExecuteQueueRuntimeMBean`, which returns the number of requests that have been processed by the corresponding execution queue. WebLogic Server uses execute queues to optimize the performance of critical applications. For more information, refer to "Using Execute Queues to Control Thread Usage."

To create a counter monitor for an `ExecuteQueueRuntimeMBean` on a single server instance, the example class in Listing 6-8:

1. Looks up the `javax.management.MBeanServer` through the server's JNDI tree.

   See "Using the MBeanServer Interface to Access MBeans" on page 2-18.

2. Constructs an object name for the monitor MBean instance.

   The object name must be unique throughout the entire WebLogic Server domain and it must follow the JMX conventions:

   domain name:Name=*name*,Type=*type*[,*attr*=*value*]...

   See the Javadoc for `javax.management.ObjectName`, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html.

3. Creates an instance of `CounterMonitorMBean` and registers it under the object name that you created in the previous step.

   The `MBeanServer.createMBean` method both creates the MBean object in the WebLogic Server JVM and registers the object in the server's MBean server.

4. Configures the monitor MBean by doing the following:

   a. Constructs the object name of the observed MBean and sets it as the value of the `CounterMonitorMBean ObservedObject` attribute.

      For a list of commonly monitored MBeans and their `WebLogicObjectName`, refer to Table 6-1, "Commonly Monitored WebLogic Server Attributes," on page 6-6.

   b. Sets values of the monitor MBean attributes.

      For information about the attributes and operations that you use to configure monitors, refer to:

      "Configuring CounterMonitor Objects" on page 6-31

      "Configuring GaugeMonitor Objects" on page 6-33

      "Configuring StringMonitor Objects" on page 6-34.

5. Instantiates the listener object that you created in "Creating a Notification Listener for a Monitor MBean" on page 6-25.

6. Registers the listener object using the MBean server's `addNotificationListener()` operation.

7. Starts the monitor using the monitor's `start()` operation.

In the example, `weblogic` is a user who has permission to view and modify MBean attributes. For information about permissions to view and modify MBeans, refer to "Security Roles" in the *Securing WebLogic Resources* guide.

Listing 6-8 Instantiating a Counter Monitor and Listener on a Single Server

```
import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.MBeanServer;
import javax.management.ObjectInstance;
import javax.management.ObjectName;
import javax.management.monitor.CounterMonitor;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class ClientMonitor {
   // The name of the WebLogic domain. Please change this to match the
   // name of your installation specific domain name
   private static String weblogicDomain = "mydomain";
   // The name and URL of the WebLogic server. Please change these to match the
   // name of your installation specific server name and URL
   private static String weblogicServer = "myserver";
   private static String url = "t3://localhost:7001";
  // The credentials for a user in the Administrator role. Please change these
  // to match the name of an administrator in your security realm.
   private static String username = "weblogic";
   private static String password = "weblogic";

   public static void main(String Args[]) {
      try {
         //Get the MBeanServer interface this is needed when you are
         // creating/registering a monitor from the client side.
         MBeanServer rmbs = null;
         Hashtable props = new Hashtable();
         props.put(Context.PROVIDER_URL, url);
         props.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
         props.put(Context.SECURITY_PRINCIPAL, username);
         props.put(Context.SECURITY_CREDENTIALS, password);
         InitialContext ctx = new InitialContext(props);
         rmbs = (MBeanServer) ctx.lookup("weblogic.management.server");

         // Construct the objectName for your CounterMonitor object
         ObjectName monitorObjectName = new ObjectName(
            "mcompany:Name=MyCounter,Type=CounterMonitor");
```

```
    // Create the Monitor MBean.
    rmbs.createMBean(
       "javax.management.monitor.CounterMonitor", monitorObjectName);

    //Configure your monitor object using the MBean attributes
    AttributeList monitorAttributes = new AttributeList();
    // Construct the objectName for the observed MBean
    ObjectName qObjectName = new ObjectName(weblogicDomain
       + ":Name=weblogic.kernel.Default,Location=" + weblogicServer
       + ",Type=ExecuteQueueRuntime,ServerRuntime=" + weblogicServer);
    Attribute observedObjectAttribute = new Attribute("ObservedObject",
       qObjectName);
    monitorAttributes.add(observedObjectAttribute);

    Attribute observedAttributeAttribute =
       new Attribute("ObservedAttribute", "ServicedRequestTotalCount");
    monitorAttributes.add(observedAttributeAttribute);

    Attribute notifyAttribute = new Attribute("Notify", new Boolean(true));
    monitorAttributes.add(notifyAttribute);

    // Define variables to be used when configuring your
    // CounterMonitor object.
    Integer threshold = new Integer(10);
    Integer offset = new Integer(1);
    Attribute thresholdAttribute = new Attribute("Threshold", threshold);
    monitorAttributes.add(thresholdAttribute);

    Attribute offsetAttribute = new Attribute("Offset", offset);
    monitorAttributes.add(offsetAttribute);

    monitorAttributes = rmbs.setAttributes(monitorObjectName,
       monitorAttributes);

    //Instantiate and register your listener with the monitor
    CounterListener listener = new CounterListener();
    rmbs.addNotificationListener(monitorObjectName, listener, null, null);

    // Start the monitor
    Object[] params = new Object[0];
    String[] signature = new String[0];
    rmbs.invoke(monitorObjectName, "start", params, signature);

     // Prevent the client program from exiting
    synchronized (listener) {
       try {
          listener.wait();
       } catch (InterruptedException ignore) {
       }
    }
} catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Example: Monitoring Instances of an MBean on Multiple Servers

A WebLogic Server domain maintains a set of MBean instances for each server instance. For example, each server instance hosts its own `ServerRuntimeMBean`, `LogMBean`, and `ExecuteQueueRuntimeMBean`. As a convenience, you can access all of these MBean instances from a single connection to the Administration Server. This single connection also enables you to create monitor MBeans for these MBeans on all servers in the domain. For example, your JMX client can connect to the Administration Server and create a counter monitor MBean on each Managed Server to monitor the server's `ExecuteQueueRuntimeMBean`.

JMX clients that use this technique must import the `weblogic.management.MBeanHome` and `weblogic.management.runtime.ServerRuntimeMBean` classes.

To monitor instances of `ExecuteQueueRuntimeMBean` on each server instance in a domain, the code excerpt in Listing 6-9 does the following:

1. Retrieves the domain's Administration `MBeanHome`.

   The Administration `MBeanHome` interface enables your client to access all active server instances in the domain without having to determine the listen address and listen port for each server instance and without having to determine whether the server instance is currently active.

2. Invokes `MBeanHome.getMBeansByType` to retrieve all instances of `ServerRuntimeMBean` in the domain.

   Only servers that are currently running maintain a `ServerRuntimeMBean` instance. If a server is not running, the `MBeanHome.getMBeansByType` method will not return a `ServerRuntimeMBean` for the server.

3. For each `ServerRuntimeMBean`, the code does the following:

   a. Gets the corresponding server's local `MBeanHome` interface from the Administration Server's JNDI tree.

      The Administration Server's JNDI tree contains a reference to local `MBeanHome` interface for each Managed Server. The `ServerRuntimeMBean.getName()` method

returns the name of the server instance. This returned value, plus `MBeanHome.JNDI_NAME` is the JNDI name of the server's local MBeanHome interface.

b. Uses the `MBeanHome.getMBeanSever()` method to get the local server's `javax.management.MBeanServer` interface.

Once the code retrieves the local server's `MBeanServer` interface, it can proceed with creating monitor MBeans as in Listing 6-8.

Listing 6-9 Instantiating a Monitor on Multiple Server Instances

```
import weblogic.management.MBeanHome;
import weblogic.management.runtime.ServerRuntimeMBean;
...
// Get the Admin home
MBeanHome adminhome = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
// Get the list of running managed servers and iterate over it
Set srSet = adminhome.getMBeansByType("ServerRuntime");
Iterator sr_iter = srSet.iterator();
while (sr_iter.hasNext()) {
   ServerRuntimeMBean bean = (ServerRuntimeMBean) sr_iter.next();
   // Get the local home for the managed server
   MBeanHome localhome =
      (MBeanHome) ctx.lookup(MBeanHome.JNDI_NAME + "." + bean.getName());
   // Get the MBeanServer for the managed server
   MBeanServer rmbs = localhome.getMBeanServer();
... // code to create monitor MBeans
}
```

Configuring CounterMonitor Objects

`CounterMonitor` objects observe changes in MBean attributes that are expressed as integers. The following list describes groups of `CounterMonitor` attributes that you set to achieve typical configurations of a `CounterMonitor` instance:

- Sends a notification when the observed attribute exceeds the threshold.

  `Threshold`
  `Notify` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`

- Sends a notification when the observed attribute exceeds the threshold. Then it increases the threshold by the offset value. Each time the observed attribute exceeds the new threshold, the threshold is increased by the offset value. For example, if you set `Threshold` to `1000` and `Offset` to `2000`, when the observed attribute exceeds `1000`, the `CounterMonitor` object sends a notification and increases the threshold to `3000`. When the observed attribute exceeds `3000`, the `CounterMonitor` object sends a notification and increases the threshold again to `5000`.

  `Threshold`
  `Notify` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`
  `Offset`

- Sends a notification when the observed attribute exceeds the threshold, and increases the threshold by the offset value. When the threshold reaches the value specified by the `Modulus` attribute, the threshold is returned to the value that was specified through the latest call to setter for the monitor's `Threshold` attribute, before any offsets were applied. For example, if the original `Threshold` is set to `1000` and the `Modulus` is set to `5000`, when the `Threshold` exceeds `5000`, the monitor sends a notification and resets the `Threshold` to `1000`.

  `Threshold`
  `Notify` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`
  `Offset`
  `Modulus`

- Sends a notification when the difference between two consecutive observations exceeds the threshold. For example, the `Threshold` is `20` and the monitor observes an attribute value of `2`. If the next observation is greater than `22`, then the monitor sends a notification. However, if the value is `10` at the next observation, and `25` at the following observation, then the monitor does not send a notification because the value has not changed by more than `20` for any two consecutive observations.

  `Threshold`
  `Notify` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`
  `DifferenceMode` (set to `true`)

- Sends a notification when the difference between two consecutive observations exceeds the threshold, and increases the threshold by the offset value. When the threshold reaches the

value specified by the `Modulus` attribute, the threshold is returned to the value that was specified through the latest call to setter for the monitor's `Threshold` attribute, before any offsets were applied.

`Threshold`
`Notify` (set to `true`)
`ObservedObject`
`ObservedAttribute`
`Offset`
`Modulus`
`DifferenceMode` (set to `true`)

To see all possible configurations of a `CounterMonitor` instance, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

Configuring GaugeMonitor Objects

`GaugeMonitor` objects observe changes in MBean attributes that are expressed as integers or floating-point. The following list describes groups of `GaugeMonitor` attributes and operations that you use to achieve typical configurations of a `GaugeMonitor` instance:

- Sends a notification when the observed attribute is beyond the high threshold.

  Set the following attributes:

  `HighThreshold`
  `NotifyHigh` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`

- Sends a notification when the observed attribute is outside the range of the high or low threshold.

  Set the following attributes:

  `HighThreshold`
  `NotifyHigh` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`
  `NotifyLow` (set to `true`)

- Sends a notification when the difference between two consecutive observations is outside the range of the high or low threshold.

Set the following attributes:

`NotifyHigh` (set to `true`)
`ObservedObject`
`ObservedAttribute`
`NotifyLow` (set to `true`)
`DifferenceMode` (set to `true`)

Invoke the following operation as well:
`setThresholds(`*`int Highthreshold, Lowthreshold`*`)`

`GaugeMonitor` does not support an offset or modulus.

To see all possible configurations of a `GaugeMonitor` instance, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

Configuring StringMonitor Objects

`StringMonitor` objects observe changes in MBean attributes that are expressed as strings. The following list describes groups of `StringMonitor` attributes that you set to achieve typical configurations of a `StringMonitor` instance:

- Sends a notification when the observed attribute **matches** the string specified in `StringToCompare`.

  `StringToCompare`
  `NotifyMatch` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`

- Sends a notification when the observed attribute **differs from** the string specified in `StringToCompare`.

  `StringToCompare`
  `NotifyDiffer` (set to `true`)
  `ObservedObject`
  `ObservedAttribute`

To see all possible configurations of a `StringMonitor` instance, refer to the JMX 1.0 API documentation, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html. The archive that you download includes the API documentation.

CHAPTER 7

Using the WebLogic Timer Service to Generate and Receive Notifications

WebLogic Server includes a timer service that you can configure to emit notifications at specific dates and times or at a constant interval. To listen and respond to these timer notifications, you create a JMX notification listener and register it with the timer service.

The WebLogic timer service extends the standard JMX timer service, enabling it to run within a Weblogic Server execute thread and within the security context of a WebLogic Server user account. (**Execute threads** enable you to fine-tune your application's use of server resources and optimize performance. For more information, refer to "Using Execute Queues to Control Thread Usage" in *WebLogic Server Performance and Tuning*.)

The following sections describe how to use the WebLogic timer service:

Using the WebLogic Timer Service: Main Steps

WebLogic Server does not provide a centralized timer service that can be accessed by all resources that are deployed on a specific server instance. Instead, each application constructs and manages instances of the timer service as it requires. Each time you restart a server instance, each application must re-instantiate any timer service configurations it needs.

To configure the WebLogic timer service to emit notifications and to create a listener that receives those notifications, create a class that does the following:

1. Constructs an instance of the `weblogic.management.timer.Timer` MBean.
2. Invokes the `Timer.addNotification` API to configure the `Timer` MBean to emit a notification object at a specific time or at a recurring interval.

   The class can invoke the `addNotification` API multiple times to configure the `Timer` MBean to emit notification objects at different times and time intervals.

   See "Configuring a Timer MBean to Emit Notifications" on page 7-2.
3. Creates a notification listener with optional filter and registers the listener and filter with the `Timer` MBean.

   Your class can register multiple listeners and filters with the `Timer` MBean instance.

   For information about creating and registering listeners, refer to "Listening for Notifications from WebLogic Server MBeans: Main Steps" on page 6-9.
4. Invokes the `Timer.start` method to start an instance of the timer service.

When your listener receives notifications, it can invoke `TimerNotification` methods to retrieve data from the notification.

An application can include multiple classes that construct and configure a `Timer` MBean. Each class uses its own instance of the `Timer` MBean and listens for notifications only from the `Timer` MBean that it instantiated.

Configuring a Timer MBean to Emit Notifications

To configure a `Timer` MBean instance to emit notifications, you invoke the MBean's `addNotification` method. The method includes parameters that configure the frequency of notifications and specify a handback object.

When you invoke the `addNotification` method, the timer service creates a `TimerNotification` object and returns an identifier for the new object. You can use this identifier to retrieve information about the `TimerNotification` object from the timer or to remove the object from the timer's list of notifications. When the time that you specify arrives, the timer service emits the `TimerNotification` object along with a reference to the handback object.

The method signature for `addNotification` is as follows:

```
addNotification (java.lang.String type, java.lang.String message,
                 java.lang.Object userData,java.util.Date startTime,
                 long period, long nbOccurences)
```

Table 7-1 describes each parameter of the `addNotification` API. For more information, refer to the WebLogic Server Javadoc for `weblogic.management.timer.Timer`.

Table 7-1 Parameters of the addNotification API

Parameter	Description
`java.lang.String type`	A string that you use to identify the event that triggers this notification to be broadcast. For example, you can specify `midnight` for a notification that you configure to be broadcast each day at midnight.
`java.lang.String message`	Specifies the value of the `TimerNotification` object's `message` attribute.
`java.lang.Object userData`	Specifies the name of an object that contains whatever data you want to send to your listeners. Usually, you specify a reference to the class that registered the notification, which functions as a callback.
`java.util.Date startTime`	Specifies a `Date` object that contains the time and day at which the timer emits your notification. For more information, refer to the next section, "Specifying Time Intervals" on page 7-4.
`long period`	(Optional) Specifies the interval in milliseconds between notification occurrences. Repeating notifications are not enabled if this parameter is zero or is not defined (`null`). For more information, refer to the next section, "Specifying Time Intervals" on page 7-4.
`long nbOccurences`	(Optional) Specifies the total number of times that the notification will occur. If the value of this parameter is zero or is not defined (`null`) and if the period is not zero or `null`, then the notification will repeat indefinitely. If you specify this parameter, each time the `Timer` MBean emits the associated notification, it decrements the number of occurrences by one. You can use `Timer.getNbOccurrences` method to determine the number of occurrences that remain. When the number of occurrences reaches zero, the `Timer` MBean removes the notification from its list of configured notifications.

Specifying Time Intervals

To facilitate specifying dates, the `Timer` MBean includes the following integer constants:

- `ONE_SECOND`, which resolves to the number of milliseconds in one second.
- `ONE_MINUTE`, which resolves to the number of milliseconds in one minute.
- `ONE_HOUR`, which resolves to the number of milliseconds in one hour.
- `ONE_DAY`, which resolves to the number of milliseconds in one day.
- `ONE_WEEK`, which resolves to the number of milliseconds in one week.

For example, the following code configures the timer service to emit a `TimerNotification` object once a day at midnight:

```
java.util.Date midnight =
      java.util.Calendar.getInstance.set(HOUR_OF_DAY=24:00:00).getTime();
addNotification (eachMidnight, "the time is midnight",
      this,midnight,Timer.ONE_DAY);
```

If the time and date that you specify is earlier than the current time and date, the `addNotification` method attempts to update this entry as follows:

- If you provided a value for the `period` parameter, the method increments the `date` value by the `period` value until the date is later than the current date. For example, if you specified `ONE_DAY` for the `period` value, the `addNotification` increments the `date` value one day until it is later than the current date.
- If you provided a value for the `nbOccurrences` parameter, the method updates the notification date as explained above. Each time it increments the `date` value, it decreases the specified number of occurrences by one. If the number of occurrences reaches `0` and the notification date remains earlier than the current date, the method throws `IllegalArgumentException`.
- If you did not provide a value for the `period` parameter, the notification date cannot be updated and the method throws `IllegalArgumentException`.

Example: Generating a Notification Every Minute

The code in Listing 7-1 is a servlet listener that configures the timer service to emit notifications every minute. It takes the following actions:

1. Extends the `NotificationListener` class so it can listen for notifications from the `Timer` MBean that the class instantiates.

   The `handleNotification` method that all listeners must implement is at the end of the class.

2. Instantiates a `weblogic.management.timer.Timer` MBean.

3. Registers itself with the `Timer` MBean as a notification listener.

4. Invokes the `addNotification` method.

   The Date object configures the timer to start emitting this notification 5 seconds after it is added to the `Timer` MBean's notification list. The `PERIOD` value causes the notification to be emitted every minute.

   The class attaches itself as the user-defined `userData` object.

   Each time the `Timer` MBean emits a notification, it will emit a `TimerNotification` object that contains this object, and whose `Message` attribute contains the string `a recurring call`, and whose `Type` attribute contains the string `oneMinuteTimer`.

5. Starts the `Timer` MBean instance.

When you redeploy or undeploy the servlet, it invokes it the `Timer.stop` method for the `Timer` MBean instance that is represented by the `timer` variable.

Listing 7-1 Servlet Listener

```
import java.util.Date;

import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.InstanceNotFoundException;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import weblogic.management.timer.Timer;

// Implementing NotificationListener
public final class LifecycleListener implements ServletContextListener,
               NotificationListener {

    private static final long PERIOD = Timer.ONE_MINUTE;
    private Timer timer;
    private Integer notificationId;
```

```
    public void contextInitialized(ServletContextEvent event) {
        System.out.println(">>> contextInitialized called.");

        // Instantiating the Timer MBean
        timer = new Timer();

        // Registering this class as a listener
        timer.addNotificationListener(this, null, "some handback object");

        // Adding the notification to the Timer and assigning the
        // ID that the Timer returns to a variable
        Date timerTriggerAt = new Date((new Date()).getTime() + 5000L);
        notificationId = timer.addNotification("oneMinuteTimer",
                         "a recurring call", this,
                         timerTriggerAt, PERIOD);
        timer.start();
        System.out.println(">>> timer started.");
    }

    public void contextDestroyed(ServletContextEvent event) {
        System.out.println(">>> contextDestroyed called.");
        try {
            timer.stop();
            timer.removeNotification(notificationId);
            System.out.println(">>> timer stopped.");
       } catch (InstanceNotFoundException e) {
            e.printStackTrace();
        }
    }
    /* callback method */
    public void handleNotification(Notification notif, Object handback) {
            System.out.println(">>> "+(new Date())+
                         " timer handleNotification="+notif+
                         ", handback="+handback);
    }
}
```

The deployment descriptor for the servlet listener must use the `<listener>` and `<listener-class>` statements to declare the class in the example above. See Listing 7-2.

Listing 7-2 Deployment Descriptor for Servlet Listener

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
```

```
    <listener>
        <listener-class>
            examples.jmxtimer.LifecycleListener
        </listener-class>
    </listener>
</web-app>
```

Removing Notifications

The `Timer` MBean removes notifications from its list when either of the following occurs:

- A non-repeating notification has been emitted.
- A repeating notification has exhausted its number of occurrences.

The `Timer` MBean also provides the following APIs to remove notifications:

- `removeAllNotifications()`, which remove all notifications that are registered with the `Timer` MBean instance.
- `removeNotification(java.lang.Integer id)`, which removes the notification whose ID matches the ID you specify. The `addNotification` method returns this ID when you invoke it. You can also use `Timer` APIs to retrieve IDs.
- `removeNotifications(java.lang.String type)`, which removes all notifications whose type corresponds to the type that you specify.

To use these remove notification APIs for a given `Timer` MBean instance, add them to the class that you use to instantiate the `Timer` MBean. Wrap each API within a method, similar to the `timer.start()` and `timer.stop()` invocations in Listing 7-1. For example, if you assigned the `Timer` MBean instance to a variable named `timer`, add the following method to your class:

```
public void removeMyNotification (java.lang.Integer id) {
    timer.removeNotification(id);
}
```

For more information, refer to the WebLogic Server Javadoc for `weblogic.management.timer.Timer`.

Index

N

O

P

R