BEA WebLogic Server®

Developing Custom Management Utilities with JMX

Version 9.0
Revised: February 23, 2007

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Understanding WebLogic Server MBeans

6. Managing Security Realms with JMX

7. Using Notifications and Monitor MBeans

8. Configuring WebLogic Server JMX Services

CHAPTER 1

Introduction and Roadmap

To integrate third-party management systems with the WebLogic Server management system, WebLogic Server provides standards-based interfaces that are fully compliant with the Java Management Extensions (JMX) specification. Software vendors can use these interfaces to monitor WebLogic Server MBeans, to change the configuration of a WebLogic Server domain, and to and monitor the distribution (activation) of those changes to all server instances in the domain. While JMX clients can perform all WebLogic Server management functions without using BEA's proprietary classes, BEA recommends that remote JMX clients use WebLogic Server protocols (such as T3) to connect to WebLogic Server instances.

This document describes creating JMX clients that monitor and modify WebLogic Server resources.

The following sections describe the contents and organization of this guide—*Developing Custom Management Utilities with JMX*.

Document Scope and Audience

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

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

Guide to this Document

- This chapter, Introduction and Roadmap, introduces the organization of this guide.
- Chapter 2, "Understanding WebLogic Server MBeans," describes the JMX services that you use to monitor and manage WebLogic Server MBeans and introduces the data model that organizes WebLogic Server MBeans.
- Chapter 3, "Overview of WebLogic Server Subsystem MBeans," introduces the MBeans that can be used to monitor and manage various subsystems of WebLogic Server.
- Chapter 4, "Accessing WebLogic Server MBeans with JMX," provides instructions and examples for accessing WebLogic Server MBeans from a JMX client.
- Chapter 5, "Managing a Domain's Configuration with JMX," provides instructions and examples for managing a WebLogic Server domain's configuration through JMX.
- Chapter 7, "Using Notifications and Monitor MBeans," describes working with notifications and listeners to listen for changes in WebLogic Server MBean attributes.
- Chapter 8, "Configuring WebLogic Server JMX Services," describes how to specify which JMX services are available in a domain.

Related Documentation

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

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

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

You can view the API reference for the `javax.management*` packages from: http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html.

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

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

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

- *Developing WebLogic Server Applications* is a guide to developing WebLogic Server applications.
- *Developing Manageable Applications with JMX* describes how to create and register custom MBeans.

New and Changed JMX Features in This Release

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

JMX 1.2 and JMX Remote API 1.0 (JSR-160)

WebLogic Server uses the Java Management Extensions (JMX) 1.2 implementation that is included in JDK 1.5. Prior to WebLogic Server 9.0, WebLogic Server used its own JMX implementation based on the JMX 1.0 specification.

Remote JMX clients can use the standard JMX remote API 1.0 (JSR-160) to connect to the JMX agents in WebLogic Server. (See http://jcp.org/en/jsr/detail?id=160.) Prior to 9.0, the JMX remote API had not been published and remote JMX clients had to use BEA's proprietary APIs to connect to WebLogic Server.

Deprecated MBeanHome and Type-Safe Interfaces

Now that the JMX remote APIs (JSR-160) are published, BEA's proprietary API for remote JMX access, `weblogic.management.MBeanHome`, is no longer needed and is therefore deprecated.

The `MBeanHome` API also made it possible for BEA to provide a typed API layer over its JMX layer that you could use to interact with WebLogic Server MBeans. Your JMX application classes could import type-safe interfaces for WebLogic Server MBeans; retrieve a reference to the MBeans through the `weblogic.management.MBeanHome` interface; and invoke the MBean methods directly.

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

If any of your classes import the type-safe interfaces (which are under `weblogic.management`), BEA recommends that you update to using the standard JMX programming model. See Accessing WebLogic Server MBeans with JMX in *Developing Custom Management Utilities with JMX*. If you do not update your JMX clients, they will use the domain's compatibility MBean server, which is enabled by default.

If you were using the `MBeanHome` API to automate common configuration tasks, consider using the new WebLogic Scripting Tool (WLST) instead of JMX. WLST is a command-line scripting interface that manages and monitors active or inactive WebLogic Server domains. The WLST scripting environment is based on the Java scripting interpreter Jython. In addition to WebLogic scripting functions, you can use common features of interpreted languages, including local variables, conditional variables, and flow control statements. You can extend the WebLogic

scripting language by following the Jython language syntax. See http://www.jython.org. For more information on WLST, see *WebLogic Scripting Tool*.

Changes to the Model for Distributing Configuration Data in a Domain

You can now collect modifications to a domain configuration and distribute them as a group throughout the domain. This release also contains APIs that you can use to monitor the distribution of changes.

The Administration Server hosts a set of pending MBeans, which are the in-memory representation of all pending changes to a domain's configuration (pending MBean data is backed up in a pending `config.xml` file). Changes in pending MBeans do not take effect immediately. You must explicitly distribute them in a process that resembles a transaction. If any Managed Server is unable to consume a change, the entire set of changes in a distribution process is rolled back. This transactional process is the only way to change a domain's configuration through JMX. See Managing a Domain's Configuration with JMX in *Developing Custom Management Utilities with JMX*.

Prior to WebLogic Server 9.0, the Administration Server hosted a set of MBeans (administration MBeans) that represented the persisted configuration for all servers and server resources in a domain. To enhance performance, each server instance replicated these MBeans locally and used the replicas, called local configuration MBeans. When a JMX client changed an administration MBean, the Administration Server immediately updated the local configuration MBeans on all server instances in the domain even if the server itself could not integrate the change. In some cases, a local configuration MBean could not be updated unless you restarted a server instance, and the replica and its master administration MBean would contain different values. In addition, JMX clients could directly access local configuration MBeans and change their values, which also resulted in an inconsistent state between replica and master MBean.

Changes to the MBean Data Model

TThe JMX specification does not impose a model for organizing MBeans. However, because the configuration of a WebLogic Server domain is specified in an XML document, WebLogic Server organizes its MBeans into a hierarchical model that reflects the XML document structure.

For example, the root of a domain's configuration document is `<domain>` and below the root are child elements such as `<server>` and `<cluster>`. Each domain maintains a single MBean of type `DomainMBean` to represent the `<domain>` root element. Within `DomainMBean`, JMX

attributes provide access to the MBeans that represent child elements such as `<server>` and `<cluster>`.

Prior to WebLogic Server 9.0:

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

As of WebLogic Server 9.0:

- The data model is consistent across WebLogic Server subsystems.
- To enable JMX clients to control MBean life cycles, WebLogic Server MBeans contain operations that follow the design pattern for Java bean factory methods: for each child, a parent MBean contains a `createChild` and `destroyChild` operation, where *`Child`* is the short name of the MBean's type. (The short name is the MBean's unqualified type name without the `MBean` suffix. For example, `createServer`). The parent also contains a `lookupChild` operation and a *`Children`* attribute.

  For example, `DomainMBean` contains the `createServer`, `destroyServer`, and `lookupServer` operations and it contains a `Servers` attribute.

  There is no other option for creating child MBeans.
- JMX clients no longer need to construct JMX object names in order to retrieve a WebLogic Server MBean. Instead, they navigate the MBean hierarchy by successively invoking code similar to the following:

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

  where:

  – *`object-name`* is the object name of the current node (MBean) in the MBean hierarchy.

  – *`attribute`* is the name of an attribute in the current MBean that refers to another MBean.
- The compatibility MBean server (which you must enable if your JMX clients still use the deprecated `MBeanHome` interface) will register new instances of WebLogic Server MBeans only if the JMX client has specified a correctly constructed, hierarchical object name for the instance.

To access the hierarchy, clients can use a set of new service MBeans which are registered in an MBean server under object names that are immutable and well defined. A JMX client supplies this object name to retrieve the service MBean. Then it uses the service MBean's attributes and operations to retrieve the root of a WebLogic Server MBean hierarchy.

New Functionally Aligned MBean Servers

An Administration Server maintains three MBean servers, each of which provides access to different MBean hierarchies. The Edit MBean Server provides access to the domain's editable configuration MBeans; the Domain Runtime MBean Server provides federated access to all runtime MBeans and read-only configuration MBeans in the domain; and the Runtime MBean Server provides access only to the runtime and read-only configuration MBeans on the Administration Server.

Each Managed Server maintains a Runtime MBean Server, which provides access only to its runtime and read-only configuration MBeans.

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

See MBean Servers in *Developing Custom Management Utilities with JMX*.

Changes in Subsystem MBeans

WebLogic Server now enables application developers to create and package descriptors for the application's services (such as JMS and JDBC services) in the application EAR file. When you deploy the application, WebLogic Server creates an instance of the service and configures it as defined in the descriptor. To support these application-scoped services, many subsystems have deprecated all or part of their old JMX interfaces and replaced them with new or updated MBeans.

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

Facilities for Registering Custom MBeans

Prior to WebLogic Server 9.0, if you wanted to register custom MBeans in an MBean server on a WebLogic Server instance, you could either create your own MBean server or use `weblogic.management.RemoteMBeanServer` to register in WebLogic Server's MBean server.

As of 9.0 and JDK 1.5, you can do any of the following from a JMX client that is running in a WebLogic Server JVM:

- (Recommended) Access the Runtime MBean Server through JNDI and register custom MBeans in the Runtime MBean Server.
- Register custom MBeans in the JVM's platform MBean server.
- Create your own MBean server.

See Use the Runtime MBean Server in *Developing Manageable Applications with JMX*.

New Reference Document for WebLogic Server MBeans

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

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

CHAPTER 2

Understanding WebLogic Server MBeans

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

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

Basic Organization of a WebLogic Server Domain

A WebLogic Server administration **domain** is a collection of one or more servers and the applications and resources that are configured to run on the servers. Each domain must include a special server instance that is designated as the **Administration Server**. The simplest domain contains a single server instance that acts as both Administration Server and host for applications and resources. This domain configuration is commonly used in development environments. Domains for production environments usually contain multiple server instances (**Managed Servers**) running independently or in groups called clusters. In such environments, the Administration Server does not host production applications. For more information about

domains, refer to "Understanding WebLogic Server Domains" in *Understanding Domain Configuration*.

Separate MBean Types for Monitoring and Configuring

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

- **Runtime MBeans** contain information about the runtime state of a server and its resources. They generally contain only data about the current state of a server or resource, and they do not persist this data. When you shut down a server instance, all runtime statistics and metrics from the runtime MBeans are destroyed.
- **Configuration MBeans** contain information about the configuration of servers and resources. They represent the information that is stored in the domain's XML configuration documents.
- Configuration MBeans for system modules contain information about the configuration of services such as JDBC data sources and JMS topics that have been targeted at the system level. Instead of targeting these services at the system level, you can include services as modules within an application. These application-level resources share the life cycle and scope of the parent application. However, WebLogic Server does not provide MBeans for application modules. See Supported Deployment Units in *Deploying Applications to WebLogic Server*.

The Life Cycle of WebLogic Server MBeans

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

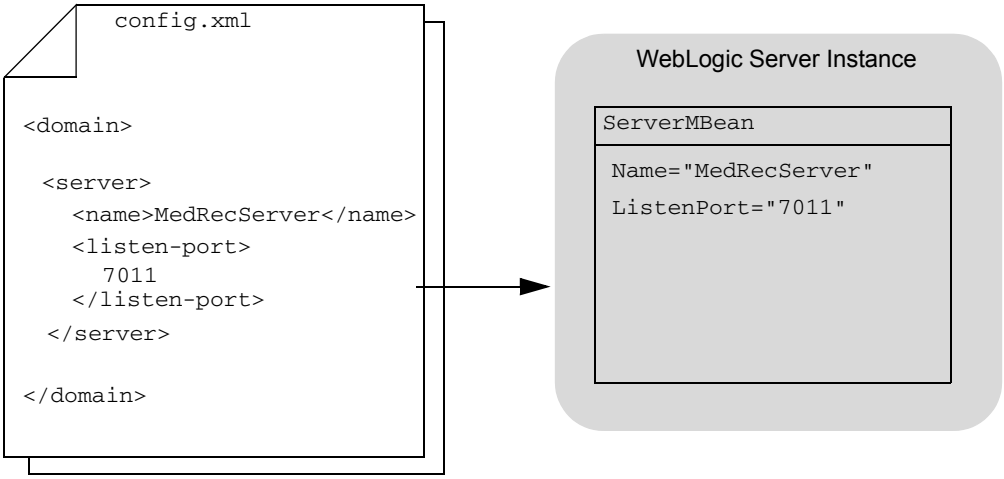
For a configuration MBean, the life cycle is as follows:

1. Each server in the domain has its own copy of the domain's configuration documents (which consist of a `config.xml` file and subsidiary files). During a server's startup cycle, it contacts the Administration Server to update its configuration files with any changes that occurred while it was shut down. Then it instantiates configuration MBeans to represent the data in the configuration documents. (See Figure 2-1.)

   Note: By default, a Managed Server will start even if it cannot contact the Administration Server to update its configuration files. This default setting creates the possibility that Managed Servers across the domain might run with inconsistent configurations. For

information about changing this default, see "Starting a Managed Server When the Administration Server Is Not Accessible" in *Managing Server Startup and Shutdown*.

Figure 2-1 Initializing Configuration MBeans on Administration Server



The configuration MBeans enable each server instance in the domain to have an identical in-memory representation of the domain's configuration.

2. To control changes to the domain's configuration, JMX clients have read-only access to these configuration MBeans.

   The Administration Server maintains a separate, editable copy of the domain's configuration documents in the domain's `config/pending` directory. It uses the data in these pending documents to instantiate a set of configuration MBeans that JMX clients can modify. After a JMX client modifies one of these configuration MBeans, the client directs the Administration Server to save the modifications in the pending configuration documents. Then the client starts a transactional process that updates the read-only configuration documents and configuration MBeans for all server instances in the domain.

   For more information, see Managing Configuration Changes in *Understanding Domain Configuration*.

3. Configuration MBeans are destroyed when you shut down the server instance that hosts them.

WebLogic Server MBean Data Model

The JMX specification does not impose a model for organizing MBeans. However, because the configuration of a WebLogic Server domain is specified in an XML document, WebLogic Server organizes its MBeans into a hierarchical model that reflects the XML document structure.

For example, the root of a domain's configuration document is `<domain>` and below the root are child elements such as `<server>` and `<cluster>`. Each domain maintains a single MBean of type `DomainMBean` to represent the `<domain>` root element. Within `DomainMBean`, JMX attributes provide access to the MBeans that represent child elements such as `<server>` and `<cluster>`.

The following sections describe the patterns that WebLogic Server MBeans use to model the underlying XML configuration:

- "Containment and Reference Relationships" on page 2-4
- "WebLogic Server MBean Object Names" on page 2-6

Containment and Reference Relationships

MBean attributes that provide access to other MBeans represent one of following types of relationships:

- Containment, which reflects a parent-child relationship between the corresponding XML elements in the domain's configuration document.
- Reference, which reflects a sibling or other non-ancestor, non-descendant relationship.

Containment Relationship

The XML excerpt in Listing 2-1 illustrates a containment relationship between `<domain>` and `<server>` and `<domain>` and `<cluster>`.

Listing 2-1 Containment Relationship in XML

```
<domain>
   <server>
      <name>MyServer</name>
   </server>
   <cluster>
```

```
      <name>MyCluster</name>
   </cluster>
</domain>
```

To reflect this relationship, `DomainMBean` has two attributes, `Servers` and `Clusters`. The value of the `Servers` attribute is an array of object names `javax.management.ObjectName[]`) for all `ServerMBeans` that have been created in the domain. The value of the `Clusters` attribute is an array of object names for all `ClusterMBeans`.

Another aspect of the containment relationship is expressed in a set of MBean operations that follow the design pattern for Java bean factory methods: for each contained (child) MBean, the parent MBean provides a `create`*`Child`* and `destroy`*`Child`* operation, where *`Child`* is the short name of the MBean's type. (The short name is the MBean's unqualified type name without the `MBean` suffix. For example, `createServer`).

Note: JMX clients cannot use `javax.management.MBeanServer.create()` or `register()` to create and register instances of WebLogic Server MBeans because WebLogic Server does not make its MBean implementation classes publicly available.

If you create and register custom MBeans (MBeans you have created to manage your applications), you will have access to your own implementation files and you can use the standard `MBeanServer.create()` or `register()` methods. Custom MBeans are not part of the WebLogic Server data model and do not participate in its factory method model.

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

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

Reference Relationship

The XML excerpt in Listing 2-2 illustrates a reference relationship between `<server>` and `<cluster>`.

Listing 2-2 Reference Relationship in XML

```
<domain>
   <server>
      <name>MyServer</name>
      <cluster>MyCluster</cluster>
   </server>
   <cluster>
      <name>MyCluster</name>
   </cluster>
</domain>
```

While a server logically belongs to a cluster, the `<server>` and `<cluster>` elements in the domain's configuration file are siblings. To reflect this relationship, `ServerMBean` has a `Cluster` attribute whose value is the object name (`javax.management.ObjectName`) of the `ClusterMBean` to which the server belongs.

MBeans in a reference relationship do not provide factory methods.

WebLogic Server MBean Object Names

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

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

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

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

where:

- `com.bea:` is the JMX domain name.

  For WebLogic Server MBeans, the JMX domain is always `com.bea`. If you create custom MBeans for your applications, name them with your own JMX domain.

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

The order of the key properties is not significant, but the name must begin with `com.bea:`.

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

Table 2-1 WebLogic Server MBean Object Name Key Properties

This Key Property	Specifies
`Name=`*`name`*	The string that you provided when you created the resource that the MBean represents. For example, when you create a server, you must provide a name for the server, such as MS1. The `ServerMBean` that represents MS1 uses `Name=MS1` in its JMX object name. If you create an MBean, you must specify a value for this `Name` component that is unique amongst all other MBeans in a domain.
`Type=`*`type`*	For configuration MBeans and runtime MBeans, the short name of the MBean's type. The short name is the unqualified type name without the `MBean` suffix. For example, for an MBean that is an instance of the `ServerRuntimeMBean`, use `ServerRuntime`. For MBeans that manage services targeted at the system level, the fully qualified name of the MBean's type including any `Bean` or `MBean` suffix. For example, for an MBean that manages a system-level JDBC data source, use `weblogic.j2ee.descriptor.wl.JDBCDataSourceBean`.

Table 2-1 WebLogic Server MBean Object Name Key Properties

This Key Property	Specifies

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

`Location=`*servername*	When you access runtime MBeans or configuration MBeans through the Domain Runtime MBean Server, the MBean object names include a `Location=`*servername* key property which specifies the name of the server instance on which that MBean is located. See "MBean Servers" on page 2-8. Singleton MBeans, such as `DomainRuntimeMBean` and `ServerLifeCycleRuntimeMBean` exist only on the Administration Server and do not need to include this key property.

MBean Servers

At the core of any JMX agent is the MBean server, which acts as a container for MBeans.

The JVM for an Administration Server maintains three MBean servers provided by BEA and optionally maintains the platform MBean server, which is provided by the JDK itself. The JVM for a Managed Server maintains only one BEA MBean server and the optional platform MBean server.

Table 2-2 describes each MBean server.

Table 2-2 MBean Servers in a WebLogic Server Domain

This MBean server	Creates, registers, and provides access to...
Domain Runtime MBean Server	MBeans for domain-wide services. This MBean server also acts as a single point of access for MBeans that reside on Managed Servers. If your JMX client accesses WebLogic Server MBeans in this MBean server by constructing object names, the client must add a `Location=`*`servername`* key property to the MBean object name. See "WebLogic Server MBean Object Names" on page 2-6. Only the Administration Server hosts an instance of this MBean server.
Runtime MBean Server	MBeans that expose monitoring, runtime control, and the active configuration of a specific WebLogic Server instance. You can also register your own (custom) MBeans in this MBean server (see Use the Runtime MBean Server in *Developing Manageable Applications with JMX*). Each server in the domain hosts an instance of this MBean server.

Table 2-2 MBean Servers in a WebLogic Server Domain

This MBean server	Creates, registers, and provides access to...
Edit MBean Server	Pending configuration MBeans and operations that control the configuration of a WebLogic Server domain. It exposes a `ConfigurationManagerMBean` for locking, saving, and activating changes. Only the Administration Server hosts an instance of this MBean server.
The JVM's platform MBean server	MBeans provided by the JDK that contain monitoring information for the JVM itself. You can register custom MBeans in this MBean server, but BEA recommends that you register them in its Runtime MBean Server. You can also configure the WebLogic Server Runtime MBean Server to be the platform MBean server, in which case the platform MBean server provides access to JVM MBeans, Runtime MBeans, and active configuration MBeans that are on a single server instance. See Using the JVM Platform MBean Server in *Developing Manageable Applications with JMX*. **Note:** Remote access to the platform MBean server can be secured only by standard JDK 1.5 security features (see http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote). If you have configured the WebLogic Server Runtime MBean Server to be the platform MBean server, enabling remote access to the platform MBean server creates an access path to WebLogic Server MBeans that is not secured through the WebLogic Server security framework.

Connecting to MBean Servers

JMX enables both local and remote access to MBean servers, but JMX clients use different APIs for the two types of access and WebLogic Server MBean servers expose different capabilities to local clients and remote clients.

Local Connections to MBean Servers

JMX clients running within a WebLogic Server JVM can access the server's Runtime MBean Server directly through JNDI and must be authenticated to do so. This is the only WebLogic Server MBean server that allows local access. When accessed from a local client, the Runtime MBean Server returns its `javax.management.MBeanServer` interface, which enables clients to

access WebLogic Server Means and to create, register, and access custom MBeans. See "Make Local Connections to the Runtime MBean Server" on page 4-7.

JMX clients can also access the local JVM's platform MBean server. The WebLogic Server security framework does not control access to the platform MBean server. Any local client can access the MBeans in this MBean server. See Using the JVM Platform MBean Server in *Developing Manageable Applications with JMX*.

Remote Connections to MBean Servers

Remote JMX clients (clients running in a different JVM from the MBean server) can use the `javax.management.remote` APIs to access any WebLogic MBean server. Clients must authenticate through the WebLogic Server security framework to do so. When accessed from a remote client, a WebLogic Server MBean server returns its `javax.management.MBeanServerConnection` interface, which enables clients only to access MBeans; remote clients cannot create and register custom MBeans. See "Make Remote Connections to an MBean Server" on page 4-2.

You can enable remote access to the platform MBean server, but such access is not secured by the WebLogic Server security framework; instead, you must use standard JDK 1.5 security features. See http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote. If it is essential that remote JMX clients have access to the JVM MBeans in the platform MBean server, see Using the JVM Platform MBean Server in *Developing Manageable Applications with JMX*.

Service MBeans

Within each MBean server, WebLogic Server registers a service MBean under a simple object name. The attributes and operations in this MBean serve as your entry point into the WebLogic Server MBean hierarchies and enable JMX clients to navigate to all WebLogic Server MBeans in an MBean server after supplying only a single object name. See Table 2-3.

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

Table 2-3 Service MBeans

MBean Server	Service MBean	JMX object name:
The Domain Runtime MBean Server	`DomainRuntimeServiceMBean` Provides access to MBeans for domain-wide services such as application deployment, JMS servers, and JDBC data sources. It also is a single point for accessing the hierarchies of all runtime MBeans and all active configuration MBeans for all servers in the domain. See `DomainRuntimeServiceMBean` in *WebLogic Server MBean Reference*.	`com.bea:Name=DomainRuntimeService, Type=weblogic.management.mbeanservers.domainruntime.DomainRuntimeServiceMBean`
Runtime MBean Servers	`RuntimeServiceMBean` Provides access to runtime MBeans and active configuration MBeans for the current server. See `RuntimeServiceMBean` in *WebLogic Server MBean Reference*.	`com.bea:Name=RuntimeService, Type=weblogic.management.mbeanservers.runtime.RuntimeServiceMBean`
The Edit MBean Server	`EditServiceMBean` Provides the entry point for managing the configuration of the current WebLogic Server domain. See `EditServiceMBean` in *WebLogic Server MBean Reference*.	`com.bea:Name=EditService, Type=weblogic.management.mbeanservers.edit.EditServiceMBean`

CHAPTER 3

Overview of WebLogic Server Subsystem MBeans

The following sections describe the MBeans that can be used to manage various subsystems of WebLogic Server:

- "Domain and Server Logging Configuration" on page 3-1
- "JMS Server and JMS System Module Configuration" on page 3-5
- "JDBC Resource Configuration" on page 3-11

In addition, for a description of MBeans that can be used to manage WebLogic Security, see "Understanding the Hierarchy of Security MBeans" on page 6-1.

Domain and Server Logging Configuration

Within a WebLogic Server domain, several MBeans configure logging services. Table 3-1 introduces the MBeans and Figure 3-1 illustrates where the MBeans are located in the configuration MBean hierarchy.

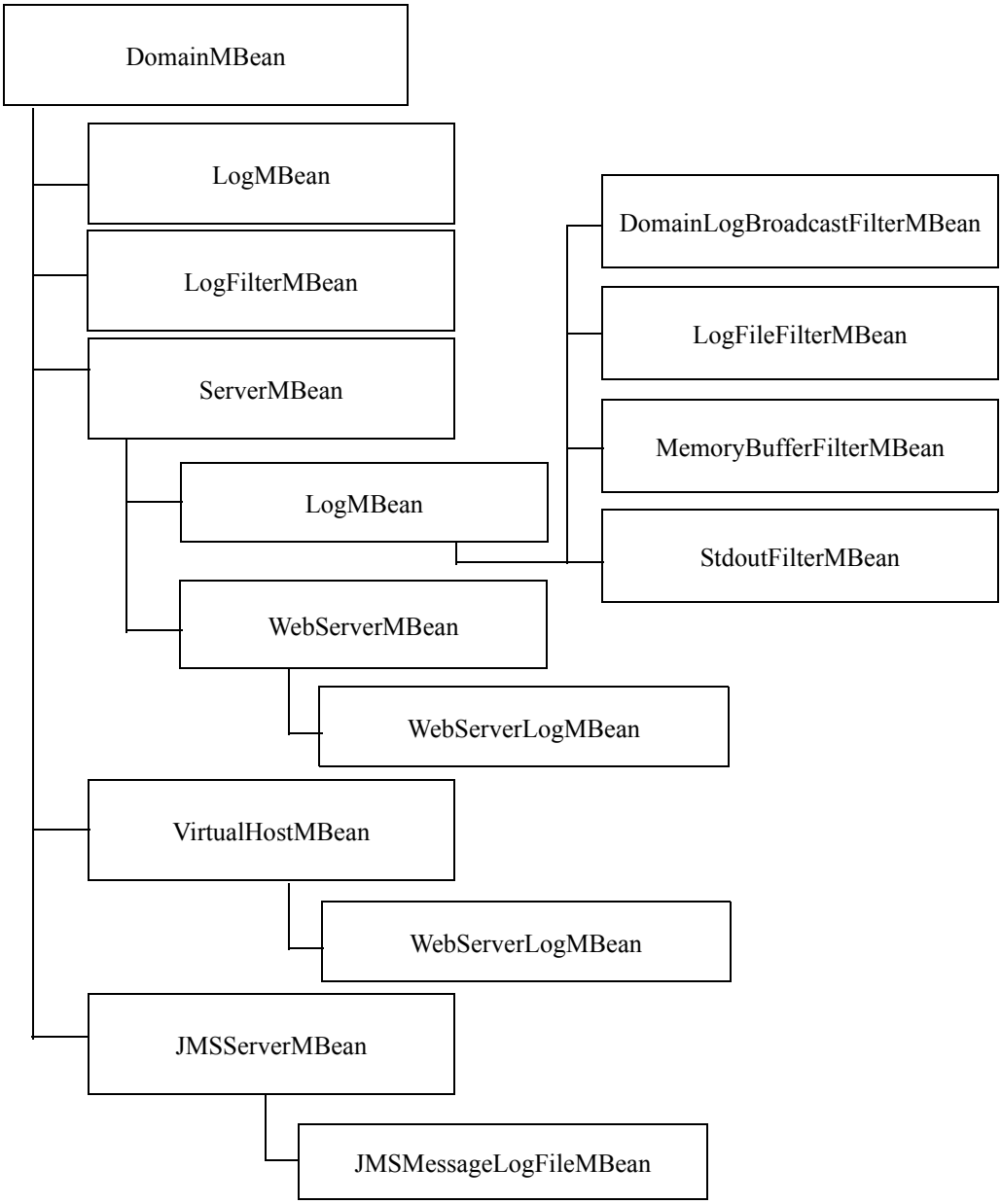
Table 3-1 MBeans for Domain and Server Logging

This MBean...	Configures...
`LogMBean`	• Threshold severity level and filter settings for logging output. • Whether the server logging is based on a Log4j implementation or the default Java Logging APIs. • Whether to redirect the JVM stdout and stderr output to the registered log destinations. 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` in the *WebLogic Server MBean Reference*.
`LogFileMBean`	Log file names and the location, file-rotation criteria, and number of files that a WebLogic Server instance uses to store log messages. See `LogFileMBean` in the *WebLogic Server MBean Reference*.
`LogFilterMBean`	A log filter which determines which messages a server instance sends to the registered log destinations. Each log filter is represented by its own instance of `LogFilterMBean`. A log filter can be defined at the domain or server level. See `LogFilterMBean` in the *WebLogic Server MBean Reference*.
`ServerMBean`	Path prefix for the server's JTA transaction log files. See `ServerMBean` in the *WebLogic Server MBean Reference*.
`WebServerMBean`	Logging HTTP requests. See `WebServerMBean` in the *WebLogic Server MBean Reference*.

Table 3-1 MBeans for Domain and Server Logging

This MBean...	Configures...
`VirtualHostMBean`	Logging HTTP requests for virtual hosts that you define. See `VirtualHostMBean` in the *WebLogic Server MBean Reference*.
`JMSServerMBean`	Message log file for this JMS Server. See `JMSServerMBean` in the *WebLogic Server MBean Reference*.

Figure 3-1 Logging MBeans



JMS Server and JMS System Module Configuration

Within a WebLogic Server domain, multiple MBeans configure JMS servers and JMS system module resources. As in prior releases, JMS servers are persisted in the domain's config.xml file and multiple JMS servers can be configured on the various WebLogic Server instances in a cluster, as long as they are uniquely named. When a JMS system module is created using JMX, WebLogic Server creates a JMS system module descriptor file in the `config\jms` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file as a `JMSSystemResource` element. This reference includes the path to the JMS system module file and a list of target servers and clusters on which the system module is deployed.

Table 3-2 introduces the MBeans and Figure 3-2 illustrates where the MBeans are located in the configuration MBean hierarchy.

Table 3-2 MBeans for JMS Servers and JMS System Module Resources

This MBean...	Configures...
`JMSServerMBean`	A JMS server is configuration entity that acts as a management container for targeted destination resources (queues and topics) in a JMS system module. A JMS server's primary responsibility for its destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. As a container for targeted destinations, any configuration or run-time changes to a JMS server can affect all of its destinations. See `JMSServerMBean` in the *WebLogic Server MBean Reference*.
`JMSSystemResourceMBean`	A JMS system resource is a resource whose definition is part of the system configuration rather than an application. The descriptor for the resource is linked through the WebLogic configuration file, but resides in a separate descriptor file. See `JMSSystemResourceMBean` in the *WebLogic Server MBean Reference*.

Table 3-2 MBeans for JMS Servers and JMS System Module Resources

This MBean...	Configures...
`SubDeploymentMBean`	Subdeployments enable administrators to deploy some resources in a JMS module to a JMS server and other JMS resources to a server instance or cluster. Standalone queues or topics can only be targeted to a single JMS server. Whereas, connection factories, uniform distributed destinations (UDDs), and foreign servers can be targeted to one or more JMS servers, one or more server instances, or to a cluster. Therefore, standalone queues or topics cannot be associated with a subdeployment if other members of the subdeployment are targeted to multiple JMS servers. However, UDDs can be associated with such subdeployments since the purpose of UDDs is to distribute its members to multiple JMS servers in a domain. See `SubDeploymemtMBean` in the *WebLogic Server MBean Reference*.
`JMSBean`	The top of the JMS module bean tree. JMS modules all have a JMSBean as their root bean (a bean with no parent). See `JMSBean` in the *WebLogic Server MBean Reference*.
`DestinationKeyBean`	Defines a unique sort order that destinations can apply to arriving messages. By default messages are sorted in FIFO (first-in, first-out) order, which sorts ascending based on each message's unique JMSMessageID. However, you can configure destination key to use a different sorting scheme for a destination, such as LIFO (last-in, first-out). See `DestinationKeyBean` in the *WebLogic Server MBean Reference*.
`DistributedQueueBean`	Defines a set of queues that are distributed on multiple JMS servers, but which are accessible as a single, logical topic to JMS clients. Distributed queues can help with load balancing and distribution, and have many of the same properties as standalone queues. See `DistributedQueueBean` in the *WebLogic Server MBean Reference*.

Table 3-2 MBeans for JMS Servers and JMS System Module Resources

This MBean...	Configures...
`DistributedTopicBean`	Defines a set of topics that are distributed on multiple JMS servers, but which are accessible as a single, logical topic to JMS clients. Distributed topics can help with load balancing and distribution, and have many of the same properties as standalone topics. See `DistributedTopicBean` in the *WebLogic Server MBean Reference*.
`ForeignServerBean`	Defines foreign messaging providers or remote WebLogic Server instances that are not part of the current domain. This is useful when integrating with another vendor's JMS product, or when referencing remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree. See `ForeignServerBean` in the *WebLogic Server MBean Reference*.
`JMSConnectionFactoryBean`	Defines a set of connection configuration parameters that are used to create connections for JMS clients. Connection factories can configure properties of the connections returned to the JMS client, and also provide configurable options for default delivery, transaction, and message flow control parameters. See `JMSConnectionFactoryBean` in the *WebLogic Server MBean Reference*.
`QueueBean`	Defines a point-to-point destination type, which are used for asynchronous peer communications. A message delivered to a queue is distributed to only one consumer. Several aspects of a queue's behavior can be configured, including thresholds, logging, delivery overrides, and delivery failure options. See `QueueBean` in the *WebLogic Server MBean Reference*.
`QuotaBean`	Controls the allotment of system resources available to destinations. For example, the number of bytes a destination is allowed to store can be configured with a Quota resource. See `QuotaBean` in the *WebLogic Server MBean Reference*.

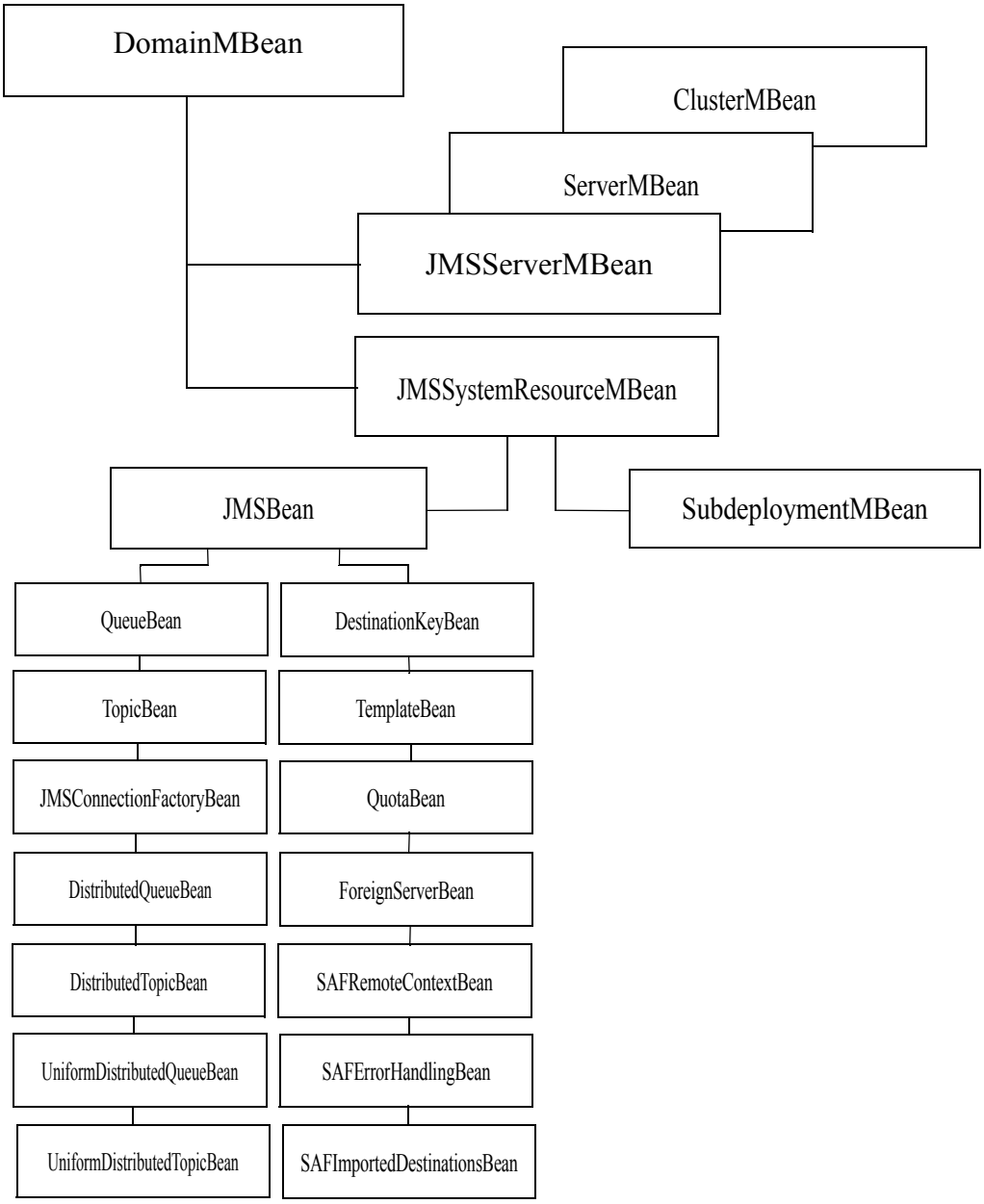
Table 3-2 MBeans for JMS Servers and JMS System Module Resources

This MBean...	Configures...
SAFRemoteContextBean	Defines the URL of the remote server instance or cluster where a JMS destination is exported from. It also contains the security credentials to be authenticated and authorized in the remote cluster or server. See SAFRemoteContextBean in the *WebLogic Server MBean Reference*.
SAFErrorHandlingBean	Defines the action to take when the SAF service fails to forward messages to remote destinations. Configuration options include an Error Handling Policy (Redirect, Log, Discard, or Always-Forward), a Log Format, and sending Retry parameters. See SAFErrorHandlingBean in the *WebLogic Server MBean Reference*.
SAFImportedDestinationsBean	Defines a collection of imported store-and-forward (SAF) destinations. A SAF destination is a representation of a queue or topic in a remote server instance or cluster that is imported into the local cluster or server instance, so that the local server instance or cluster can send messages to the remote server instance or cluster. All JMS destinations are automatically exported by default, unless the Export SAF Destination parameter on a destination is explicitly disabled. Each collection of SAF imported destinations is associated with a remote SAF context resource, and, optionally, a SAF error handling resource. See SAFImportedDestinationsBean in the *WebLogic Server MBean Reference*.
TemplateBean	Defines a set of default configuration settings for multiple destinations. If a destination specifies a template and does not explicitly set the value of a parameter, then that parameter will take its value from the specified template. See TemplateBean in the *WebLogic Server MBean Reference*.

Table 3-2 MBeans for JMS Servers and JMS System Module Resources

This MBean...	Configures...
`TopicBean`	Defines a publish/subscribe destination type, which are used for asynchronous peer communications. A message delivered to a topic is distributed to all topic consumers. Several aspects of a topic's behavior can be configured, including thresholds, logging, delivery overrides, delivery failure, and multicasting parameters. See `TopicBean` in the *WebLogic Server MBean Reference*.
`UniformDistributedQueueBean`	Defines a uniformly configured distributed queue, whose members have a consistent configuration of all distributed queue parameters, particularly in regards to weighting, security, persistence, paging, and quotas. These uniform distributed queue members are created on JMS servers based on the targeting of the uniform distributed queue. Uniform distributed queues can help with message load balancing and distribution, and have many of the same properties as standalone queues, including thresholds, logging, delivery overrides, and delivery failure parameters. See `UniformDistributedQueueBean` in the *WebLogic Server MBean Reference*.
`UniformDistributedTopicBean`	Defines a uniformly configured distributed topic, whose members have a consistent configuration of all uniform distributed queue parameters, particularly in regards to weighting, security, persistence, paging, and quotas. These uniform distributed topic members are created on JMS servers based on the targeting of the uniform distributed topic. Uniform distributed topics can help with message load balancing and distribution, and have many of the same properties as standalone topics, including thresholds, logging, delivery overrides, and delivery failure parameters. See `UniformDistributedTopicBean` in the *WebLogic Server MBean Reference*.

Figure 3-2 JMS Server and JMS System Resource MBeans



JDBC Resource Configuration

When you create a JDBC resource (data source or multi-data source) using the Administration Console or using the WebLogic Scripting Tool (WLST), WebLogic Server creates a JDBC module in the `config/jdbc` subdirectory of the domain directory, and adds a reference to the module in the domain's configuration file (`config.xml`).

Table 3-3 introduces the MBeans and Figure 3-3 illustrates where the MBeans are located in the configuration MBean hierarchy.

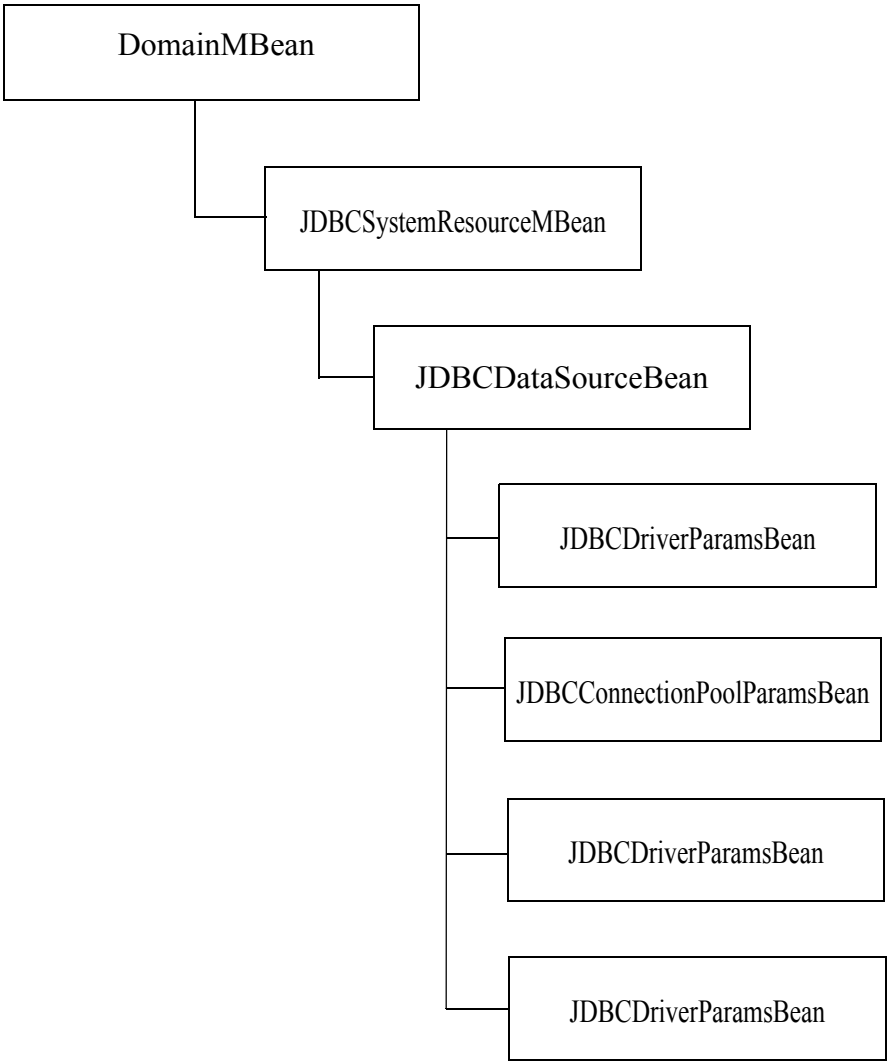
Table 3-3 MBeans for JDBC Resources

This MBean...	Configures...
`JDBCSystemResourceMBean`	A container for the JavaBeans created from a data source module. However, all JMX access for a JDBC data source is through the JDBCSystemResourceMBean. You cannot directly access the individual JavaBeans created from the data source module. See `JDBCSystemResourceMBean` in the *WebLogic Server MBean Reference*.
`JDBCDataSourceBean`	The top of the JDBC data source bean tree. JDBC data sources all have a JDBCDataSourceBean as their root bean (a bean with no parent). See `JDBCDataSourceBean` in the *WebLogic Server MBean Reference*.
`JDBCDriverParamsBean`	Contains the driver parameters of a data source. Configuration parameters for the JDBC Driver used by a data source are specified using a driver parameters bean. See `JDBCDriverParamsBean` in the *WebLogic Server MBean Reference*.
`JDBCConnectionPoolParamsBean`	Contains the connection pool parameters of a data source. Configuration parameters for a data source's connection pool are specified using the connection pool parameters bean. See `JDBCDConnectionPoolParamsBean` in the *WebLogic Server MBean Reference*.

Table 3-3 MBeans for JDBC Resources

This MBean...	Configures...
`JDBCDataSourceParamsBean`	Contains the basic usage parameters of a data source. Configuration parameters for the basic usage of a data source are specified using a data source parameters bean. See `JDBCDataSourceParamsBean` in the *WebLogic Server MBean Reference*.
`JDBCXAParamsBean`	Contains the XA-related parameters of a data source. Configuration parameters for a data source's XA-related behavior are specified using a XA parameters bean. See `JDBCXAParamsBean` in the *WebLogic Server MBean Reference*.

Figure 3-3 JDBC Resource MBeans



CHAPTER 4

Accessing WebLogic Server MBeans with JMX

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

Set Up the Classpath for Remote Clients

If your JMX client runs in its own JVM (that is, a JVM that is not a WebLogic Server instance), include the following JAR file in the client's classpath:

```
WL_HOME\lib\wljmxclient.jar
```

where `WL_HOME` is the directory in which you installed WebLogic Server.

This JAR contains BEA's implementation of the HTTP and IIOP protocols and its proprietary T3 protocol. With BEA's implementation, JMX clients send login credentials with their connection request and the WebLogic Server security framework authenticates the clients. Only authenticated clients can access MBeans that are registered in a WebLogic Server MBean server.

Note: While BEA recommends that you use its implementation of the HTTP and IIOP protocols or its proprietary T3 protocol, JMX clients can use the IIOP protocol that is defined in the standard JDK. See "Remote Connections Using Only JDK Classes" on page 4-7.

Make Remote Connections to an MBean Server

Each WebLogic Server domain includes three types of MBean servers, each of which provides access to different MBean hierarchies. See "MBean Servers" on page 2-8.

To connect to a WebLogic MBean server:

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

   Pass the following parameter values to the constructor (see `JMXServiceURL` in the *J2SE 5.0 API Specification*):

   – One of the following values as the protocol for communicating with the MBean server: `t3`, `t3s`, `http`, `https`, `iiop`, `iiops`

   – Listen address of the WebLogic Server instance that hosts the MBean server

   – Listen port of the WebLogic Server instance

   – Absolute JNDI name of the MBean server. The JNDI name must start with `/jndi/` and be followed by one of the JNDI names described in Table 4-1.

Table 4-1 JNDI Names for WebLogic MBean Servers

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

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

   The constructor method for `JMXConnector` is:

```
javax.management.remote.JMXConnectorFactory.
connector(JMXServiceURL serviceURL, Map<String,?> environment)
```

Pass the following parameter values to the constructor (see `JMXConnectorFactory` in the *J2SE 5.0 API Specification*):

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

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

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

  `javax.management.remote.JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES, "weblogic.management.remote"`

  The `weblogic.management.remote` package defines the protocols that can be used to connect to the WebLogic MBean servers. Remote JMX clients must include the classes in this package on their classpath. See "Set Up the Classpath for Remote Clients" on page 4-1.

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

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

   The `MBeanServerConnection` object is your connection to the WebLogic MBean server. You can use it for local and remote connections. See `MBeanServerConnection` in the *J2SE 5.0 API Specification*.

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

Example: Connecting to the Domain Runtime MBean Server

Note the following about the code in Listing 4-1:

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

- When the class finishes its work, it invokes `JMXConnector.close()` to close the connection to the MBean server. (See `JMXConnector` in the *J2SE 5.0 API Specification*.)

Listing 4-1 Connecting to the Domain Runtime MBean Server

```
public class MyConnection {

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

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

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

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

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

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

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

Best Practices: Choosing an MBean Server

A WebLogic Server domain maintains three types of MBean servers, each of which fulfills a specific function. Access MBeans through the MBean server that supports the task you are trying to complete:

- To modify the configuration of the domain, use the Edit MBean Server.
- To monitor changes to the pending hierarchy of configuration MBeans, use the Edit MBean Server.
- To monitor only active configuration MBeans (and not runtime MBeans), use a Runtime MBean Server.

  Monitoring through a Runtime MBean Server requires less memory and network traffic than monitoring through the Domain Runtime MBean Server. (WebLogic Server does not initialize the Domain Runtime MBean Server until a client requests a connection to it.)

  In most cases, all server instances in the domain have the same set of configuration data and it therefore does not matter whether you monitor the Runtime MBean Server on the Administration Server or on a Managed Server. However, if you make a change that a server cannot consume until it is restarted, the server will no longer accept any changes and its configuration data could become outdated. In this case, monitoring this server's Runtime MBean Server indicates only the configuration for the specific server instance. To understand the process of changing a WebLogic Server domain and activating the changes, see Managing Configuration Changes in *Understanding Domain Configuration*.
- If your client monitors runtime MBeans for multiple servers, or if your client runs in a separate JVM, BEA recommends that you connect to the Domain Runtime MBean Server on the Administration Server instead of connecting separately to each Runtime MBean Server on each server instance in the domain.

  If you register a JMX listener and filter with an MBean in the Domain Runtime MBean server, the JMX filter runs in the same JVM as the MBean it monitors. For example, if you register a filter with an MBean on a Managed Server, the filter runs on the Managed Server and forwards only messages that satisfy the filter criteria to the listener.

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

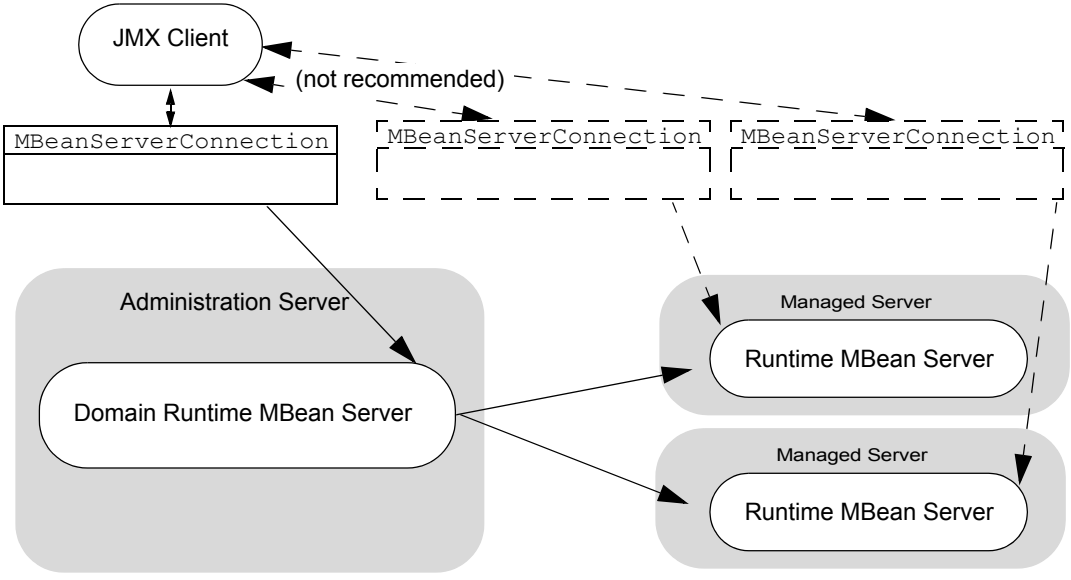
– Your code only needs to construct a single URL for connecting to the Domain Runtime MBean Server on the Administration Server. Thereafter, the code can look up values for all server instances and optionally filter the results.

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

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

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

Figure 4-1 Domain Runtime MBean Server versus Runtime MBean Server



Remote Connections Using Only JDK Classes

BEA recommends that you use WebLogic Server classes to connect from remote JMX clients. However, it is possible for remote JMX clients to connect to a WebLogic Server JMX agent using only the classes in the JDK. To do so:

1. Enable the IIOP protocol for the WebLogic Server instance that hosts your MBeans. Configure the default IIOP user to be a WebLogic Server user with Administrator privileges.

   See Enable and Configure IIOP in *Administration Console Online Help*.

2. In your JMX client, construct a `javax.management.JMXConnector` object as follows:

```
String hostname = "WLS-host"
int port = WLS-port
String protocol = "rmi";
String jndiroot= new String("/jndi/iiop://" + hostname + ":" +
       port + "/");
String mserver = "MBean-server-JNDI-name";

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

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

connector = JMXConnectorFactory.connect(serviceURL, h);
```

   where *`WLS-host`* and *`WLS-port`* are the listen address and listen port of a WebLogic Server instance and *`MBean-server-JNDI-name`* is one of the values listed in Table 4-1, "JNDI Names for WebLogic MBean Servers," on page 4-2.

Note that the hash table you create does not include the name of a protocol package. By leaving this value as null, the JMX client uses the protocol definitions from the `com.sun.jmx.remote.protocol` package, which is in the JDK.

Make Local Connections to the Runtime MBean Server

Local clients can access a WebLogic Server instance's Runtime MBean Server through the JNDI tree instead of constructing a `JMXServiceURL` object. Only the Runtime MBean Server registers itself in the JNDI tree.

When accessed from JNDI, the Runtime MBean Server returns its `javax.management.MBeanServer` interface. This interface contains all of the methods in the `MBeanServerConnection` interface plus additional methods such as `registerMBean()`, which

local process can use to register custom MBeans. (See `MBeanServer` in the *J2SE 5.0 API Specification*.)

If the classes for the JMX client are located at the top level of an enterprise application (that is, if they are deployed from the application's `APP-INF` directory), then the JNDI name for the Runtime MBean Server is:

```
java:comp/jmx/runtime
```

If the classes for the JMX client are located in a J2EE module, such as an EJB or Web application, then the JNDI name for the Runtime MBeanServer is:

```
java:comp/env/jmx/runtime
```

For example:

```
InitialContext ctx = new InitialContext();
server = (MBeanServer)ctx.lookup("java:comp/env/jmx/runtime");
```

Navigate MBean Hierarchies

WebLogic Server organizes its MBeans in a hierarchical data model. (See "WebLogic Server MBean Data Model" on page 2-4.) In this model, all parent MBeans include attributes that contain the object names of their children. You use the child's object name in standard JMX APIs to get or set values of the child MBean's attributes or invoke its methods.

To navigate the WebLogic Server MBean hierarchy:

1. Initiate a connection to an MBean server.

   See the previous section, "Make Remote Connections to an MBean Server" on page 4-2.

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

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

   – *`object-name`* is the object name of the service MBean that is registered in the MBean server. (See "Service MBeans" on page 2-11.)

     Table 2-3, "Service MBeans," on page 2-12 describes the type of service MBeans that are available in each type of MBean server.

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

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

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

Example: Printing the Name and State of Servers

The code example in Listing 4-2 connects to the Domain Runtime MBean Server and uses the `DomainRuntimeServiceMBean` to get the object name for each `ServerRuntimeMBean` in the domain. Then it retrieves and prints the value of each server's `ServerRuntimeMBean` `Name` and `State` attributes.

Note the following about the code in Listing 4-2:

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

Listing 4-2 Example: Print the Name and State of Servers

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

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

public class PrintServerState {

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

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

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

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

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

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

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

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

Example: Monitoring Servlets

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

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

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

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

The code in Listing 4-3 navigates the hierarchy described in the previous paragraphs and gets values of `ServletRuntimeMBean` attributes.

Listing 4-3 Monitoring Servlets

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

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

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

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

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

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

   /*
   * Get an array of ServerRuntimeMBeans
```

```
*/
public static ObjectName[] getServerRuntimes() throws Exception {
   return (ObjectName[]) connection.getAttribute(service,
      "ServerRuntimes");
}

/*
* Get an array of WebApplicationComponentRuntimeMBeans
*/
public void getServletData() throws Exception {
   ObjectName[] serverRT = getServerRuntimes();
   int length = (int) serverRT.length;
   for (int i = 0; i < length; i++) {
      ObjectName[] appRT =
         (ObjectName[]) connection.getAttribute(serverRT[i],
         "ApplicationRuntimes");
      int appLength = (int) appRT.length;
      for (int x = 0; x < appLength; x++) {
         System.out.println("Application name: " +
           (String)connection.getAttribute(appRT[x], "Name"));
         ObjectName[] compRT =
            (ObjectName[]) connection.getAttribute(appRT[x],
            "ComponentRuntimes");
         int compLength = (int) compRT.length;
         for (int y = 0; y < compLength; y++) {
            System.out.println("  Component name: " +
              (String)connection.getAttribute(compRT[y], "Name"));
            String componentType =
               (String) connection.getAttribute(compRT[y], "Type");
            System.out.println(componentType.toString());
            if (componentType.toString().equals("WebAppComponentRuntime")){
               ObjectName[] servletRTs = (ObjectName[])
                  connection.getAttribute(compRT[y], "Servlets");
               int servletLength = (int) servletRTs.length;
               for (int z = 0; z < servletLength; z++) {
                  System.out.println("    Servlet name: " +
                     (String)connection.getAttribute(servletRTs[z],
                      "Name"));
                  System.out.println("       Servlet context path: " +
                     (String)connection.getAttribute(servletRTs[z],
                      "ContextPath"));
                  System.out.println("       Invocation Total Count : " +
                     (Object)connection.getAttribute(servletRTs[z],
                      "InvocationTotalCount"));
               }
            }
         }
      }
```

```
      }
   }

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

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

CHAPTER 5

Managing a Domain's Configuration with JMX

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

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

Editing MBean Attributes: Main Steps

To edit MBean attributes:

1. Start an Edit Session.

   All edits to MBean attributes occur within the context of an edit session, and within each WebLogic Server domain only one edit session can be active at a time. Once a user has started an edit session, WebLogic Server locks other users from accessing the pending configuration MBean hierarchy. See "Managing Locks" on page 5-16.

2. Change Attributes or Create New MBeans.

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

3. Save Changes to the Pending Configuration Files.

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

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

4. Activate Your Saved Changes.

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

5. Restart any server instances that have been updated with changes that require a server restart.

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

Start an Edit Session

To start an edit session:

1. Initiate a connection to the Edit MBean Server.

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

   See "Make Remote Connections to an MBean Server" on page 4-2.

2. Get the object name for `ConfigurationManagerMBean`.

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

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

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

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

   where:

   - *`object-name`* is the literal `"com.bea:Name=EditService,Type=weblogic.management.mbeanservers.edit.EditServiceMBean"`, which is the object name of `EditServiceMBean`.
   - *`attribute`* is the literal `"ConfigurationManager"`, which is the name of the attribute in `EditServiceMBean` that contains `ConfigurationManagerMBean`.

3. Start an edit session.

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

   - *`waitTime`* specifies how many milliseconds `ConfigurationManagerMBean` waits to establish a lock on the edit MBean hierarchy. You cannot establish a lock if other edits are in progress unless you have administrator privileges (see "Managing Locks" on page 5-16).
   - *`timeout`* specifies how many milliseconds you have to complete your edit session. If the time expires before you save or activate your edits, all of your unsaved changes are discarded.

The `startEdit` operation returns either of the following:

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

Change Attributes or Create New MBeans

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

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

   See "Navigate MBean Hierarchies" on page 4-8.

2. To change the value of an MBean attribute, invoke the `MBeanServerConnection.setAttribute(`*`object-name`*`, `*`attribute`*`)` method where:

   - *`object-name`* is the object name of the MBean that you want to edit.
   - *`attribute`* is a `javax.management.Attribute` object, which contains the name of the MBean attribute that you want to change and its new value.

   To create an MBean, invoke the MBean's create method. For example, the factory method to create an instance of `ServerMBean` is `createServer(String `*`name`*`)` in `DomainMBean`. In *WebLogic Server MBean Reference*, each MBean describes the location of its factory methods. (See `ServerMBean`.)

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

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

   If it finds validation errors, the `validate()` operation throws an exception of type `weblogic.management.mbeanservers.edit.ValidationException`. See "Exception Types Thrown by Edit Operations" on page 5-9.

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

Save Changes to the Pending Configuration Files

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

Activate Your Saved Changes

To activate your saved changes throughout the domain:

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

   The `activate` operation returns an object name for an instance of `ActivationTaskMBean`, which contains information about the activation request. See "Listing and Undoing Changes" on page 5-9.

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

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

Example: Changing the Administration Port

The code example in Listing 5-1 changes the context path that you use to access the Administration Console for a domain. This behavior is defined by the `DomainMBean ConsoleContextPath` attribute.

Note the following about the code example:

- For information on how the class connects to the Edit MBean Server, see "Make Remote Connections to an MBean Server" on page 4-2.
- To simplify the code for learning purposes, exception handling in Listing 5-1 is minimal. See "Best Practices: Recommended Pattern for Editing and Handling Exceptions" on page 5-17.

Listing 5-1 Example: Changing the Administration Console's Context Path

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            "secureConsoleContext");
    }

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

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

       EditWLSMBeans ewb = new EditWLSMBeans();

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

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

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

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

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

Exception Types Thrown by Edit Operations

Table 5-1 describes all of the exception types that WebLogic Server can throw during edit operations. When WebLogic Server throws such an exception, the MBean server wraps the exception in `javax.management.MBeanException`. (See `MBeanException` in the *J2SE 5.0 API Specification*.)

Table 5-1 Exception Types Thrown by Edit Operations

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

Listing and Undoing Changes

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

- "List Unsaved Changes" on page 5-10
- "List Unactivated Changes" on page 5-10
- "List Changes in the Current Activation Task" on page 5-12
- "Undoing Changes" on page 5-13

WebLogic Server describes changes in a `Change` object, which is of type `javax.management.openmbean.CompositeType`. See `CompositeType` in the *J2SE 5.0 API Specification*.

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

List Unsaved Changes

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

Any unsaved changes are discarded when your edit session ends.

To list unsaved changes:

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

   Because `Change` is a `javax.management.openmbean.CompositeType`, you can also cast each item in the array as a `CompositeType` and invoke `CompositeType` methods on the change. See `CompositeType` in the *J2SE 5.0 API Specification*.

The code in Listing 5-2 creates a method that lists unsaved changes. It assumes that the calling method has already established a connection to the Edit MBean Server.

Listing 5-2 Example Method that Lists Unsaved Changes

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

List Unactivated Changes

When anyone saves changes, WebLogic Server persists the changes in the pending configuration files. The changes remain in these files, even across multiple editing sessions, unless a user who

has started an edit session invokes the `ConfigurationManagerMBean undoUnactivatedChanges()` operation, which reverts all unactivated changes from the pending files.

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

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

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

   Because `Change` is a `javax.management.openmbean.CompositeType`, you can also cast each item in the array as a `CompositeType` and invoke `CompositeType` methods on the change. See `CompositeType` in the *J2SE 5.0 API Specification*.

The code in Listing 5-3 creates a method that lists unactivated changes. It assumes that the calling method has already established a connection to the Edit MBean Server.

Listing 5-3 Example Method that Lists Unactivated Changes

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

List Changes in the Current Activation Task

When you activate changes, WebLogic Server creates an instance of `ActivationTaskMBean`, which contains one `Change` object for each change that is being activated. You can access these `ActivationTaskMBean`s from either of the following:

- The `ConfigurationManagerMBean` `activate()` method returns an object name for the `ActivationTaskMBean` that describes the current activation task.
- The `ConfigurationManagerMBean` `CompletedActivationTasks` attribute can potentially contain a list of all `ActivationTaskMBean` instances that have been created during the current Administration Server instantiation. See "Listing All Activation Tasks Stored in Memory" on page 5-14.

To list changes in the current activation task only:

1. Start an edit session.
2. Assign the output of the `activate` operation to an instance variable of type `javax.management.ObjectName`.
3. Get the value of the `ActivationTaskMBean` `Changes` attribute. and assign the output to a variable of type `Object[]`.
4. For each object in the array, invoke `Object.toString()` to output a description of the change.

   Because `Change` is a `javax.management.openmbean.CompositeType`, you can also cast each item in the array as a `CompositeType` and invoke `CompositeType` methods on the change. See `CompositeType` in the *J2SE 5.0 API Specification*.

The code in Listing 5-4 creates a method that lists all changes activated in the current editing session. It assumes that the calling method has already established a connection to the Edit MBean Server.

Listing 5-4 Example Method that Lists Changes in the Current Activation Task

```
public void activateAndList()
   throws Exception {
   ObjectName cfgMgr = (ObjectName) connection.getAttribute(service,
      "ConfigurationManager");
   ObjectName task = (ObjectName) connection.invoke(cfgMgr, "activate",
      new Object[] { new Long(120000) }, new String[] { "java.lang.Long" });
   Object[] changes = (Object[])connection.getAttribute(task, "Changes");
```

```
   int i = (int) changes.length;
   for (int i = 0; i< i; i++) {
      System.out.println("Changes activated: " + changes[i].toString());
   }
}
```

Undoing Changes

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

- `undo`

  Reverts unsaved changes.

- `undoUnactivatedChanges`

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

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

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

For example:

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

Tracking the Activation of Changes

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

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

ActivationTaskMBean instances in memory. To change the default, increase the value of the ConfigurationManagerMBean CompletedActivationTasksCount attribute.

The following sections describe working with instances of ActivationTaskMBean:

- "Listing the Status of the Current Activation Task" on page 5-14
- "Listing All Activation Tasks Stored in Memory" on page 5-14
- "Purging Completed Activation Tasks from Memory" on page 5-15

Listing the Status of the Current Activation Task

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

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

To list the status of the current activation task:

1. Start an edit session and change at least one MBean attribute.
2. Invoke the ConfigurationManagerMBean activate(long timeout) operation and assign the output to a variable of type ActivationTaskMBean.
3. Get the value of the ActivationTaskMBean State attribute.

Listing All Activation Tasks Stored in Memory

The ActivationTaskMBean that the activate operation returns describes only a single activation task. The Administration Server keeps this ActivationTaskMBean in memory until you purge it (see "Purging Completed Activation Tasks from Memory" on page 5-15) or the number of activation tasks exceeds the value of the ConfigurationManagerMBean CompletedActivationTasksCount attribute.

To access all ActivationTaskMBean instances that are currently stored in memory (see Listing 5-5):

1. Connect to the Edit MBean Server. (You do not need to start an edit session.)
2. Get the value of the ConfigurationManagerMBean CompletedActivationTasks attribute and assign the output to a variable of type Object[].

3. (Optional) For each object in the array, get and print the value of `ActivationTaskMBean` attributes such as `User` and `State`.

   See `ActivationTaskMBean` in *WebLogic Server MBean Reference*.

4. (Optional) For each object in the array, get the value of the `Changes` attribute. Invoke `Object.toString()` to output the value of the `Change` object.

Listing 5-5 Example Method that Lists All Activation Tasks in Memory

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

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

Purging Completed Activation Tasks from Memory

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

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

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

For example:

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

Managing Locks

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

The following actions remove the lock:

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

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

- The `ConfigurationManagerMBean` `stopEdit` operation succeeds.

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

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

- An edit session has been started under a user identity and another process starts an edit session under the same user identity.

  For example, if you use the Administration Console to start an edit session and shortly afterwards use the WebLogic Scripting Tool (WLST) to start an edit session under the same user identity, the WLST session will remove the lock from your Administration Console session.

  To prevent another process from starting an edit session under your user identity, get an exclusive lock by passing a `boolean` of value `true` to the `startEdit` operation. See `startEdit(waitTimeInMillis, timeOutInMillis, exclusive)` in the *WebLogic Server MBean Reference*.

All unsaved changes are lost when the lock is removed.

Best Practices: Recommended Pattern for Editing and Handling Exceptions

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

JMX agents wrap all exceptions in a generic exception of type `javax.management.MBeanException`. A JMX client can use the `MBeanException.getTargetException()` to unwrap the wrapped exception.

Consider using the following structure (see the pseudo-code in Listing 5-6):

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

  After this try block, one catch block for each of the following types of exception wrapped within `MBeanException`:

  - `EditTimedOutException`

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

  - `NotEditorException`

    This exception is thrown if the edit session times out or an administrator cancels your edit session. (See "Managing Locks" on page 5-16.)

  - `ValidationException`

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

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

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

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

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

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

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

– `NotEditorException`

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

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

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

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

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

  – `NotEditorException`

- A try block to stop the edit session.

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

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

  – `NotEditorException`

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

Listing 5-6 Code Outline for Editing and Exception Handling

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

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

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

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

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

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

// If you catch a wrapped NotEditorException, your changes were not activated
// because your edit session ended or was cancelled by an administrator.
// Throw the wrapped exception.
} catch (MBeanException e) {
   Exception targetException = e.getTargetException();
   if (targetException instanceof NotEditorException) {
      ...
      throw new MyAppEditSessionFailed(e);
   }
}

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

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

// Stop the edit session
try {
   connection.invoke(cfgMgr, "stopEdit", null, null);
   // If your activation attempt fails and you are ready to abandon
   // changes, there is no need to wait until your original timeout
```

```
   // period to expire. You can stop editing immediately
   // and you can safely ignore any wrapped NotEditorException.
} catch (MBeanException e) {
   Exception targetException = e.getTargetException();
   if (targetException instanceof NotEditorException) {
      //ignore
   }
}
...
// Output the information about the error that caused the activation to
// fail.
throw new MyAppEditSessionFailed(connection.getAttribute(task, "Error"));
```

Setting and Getting Encrypted Values

To prevent unauthorized access to sensitive data such as passwords, some attributes in WebLogic Server 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 `ServerMBean` exposes the password that is used to secure access through the IIOP protocol in an attribute named `DefaultIIOPPasswordEncrypted`. To support backwards compatibility, and to enable remote JMX clients to set passwords for WebLogic Server MBeans, each encrypted attribute provides a less secure means to encrypt and set its value.

The following sections describe how to work with encrypted attributes:

- "Set the Value of an Encrypted Attribute (Recommended Technique)" on page 5-21
- "Set the Value of an Encrypted Attribute (Compatibility Technique)" on page 5-22
- "Back Up an Encrypted Value" on page 5-23

Set the Value of an Encrypted Attribute (Recommended Technique)

To use this technique (see Listing 5-7):

1. In the same WebLogic Server JVM that hosts the MBean attribute, write a value to a byte array.

2. Pass the byte array to the `weblogic.management.EncryptionHelper.encrypt(byte[])` method and pass its return value to the `MBeanServerConnection.setAttribute` method.

   Avoid assigning the encrypted byte array to a variable because this causes the unencrypted byte array to remain in memory until it is garbage collected and the memory is reallocated.

3. Clear the original byte array using the `weblogic.management.EncryptionHelper.clear()` method.

Listing 5-7 Example: Set the Value of an Encrypted Attribute (Recommended Technique)

```
public void editDefaultIIOPPassword(ObjectName cfgRoot) throws Exception {
   // Get the ServerMBean from the DomainMBean
   ObjectName server = (ObjectName) connection.invoke(cfgRoot,
      "lookupServer", new Object[] { "myserver" },
   new String[] { "java.lang.String" });
   // Get new password from standard in. Assign it to a byte array.
   System.out.println("Enter new password and press enter: ");
   byte userinput[] = new byte[10];
   System.in.read(userinput);
   // Encrypt the byte array and set it as the encrypted
   // attribute value.
   Attribute newpassword = new Attribute("DefaultIIOPPasswordEncrypted",
      weblogic.management.EncryptionHelper.encrypt(userinput));
   connection.setAttribute(server, newpassword);
   System.out.println("New password is set to: " +
      connection.getAttribute(server, "DefaultIIOPPasswordEncrypted"));
   // Clear the byte array.
   weblogic.management.EncryptionHelper.clear(userinput);
   }
```

Set the Value of an Encrypted Attribute (Compatibility Technique)

Prior to 9.0, JMX clients used a different technique for setting encrypted values. JMX clients can continue to use this compatibility technique, and if you want to set encrypted values from a remote JMX client, this is the only technique available. The compatibility technique is less secure because it creates a `String` that contains your unencrypted password. Even though WebLogic Server converts the `String` to an encrypted byte array, the `String` will remain in memory until it is garbage collected and the memory is reallocated.

To use the compatibility technique:

1. Write a value to a `String`.
2. Pass the `String` as a parameter to the `MBeanServerConnection.setAttribute` method, but instead of setting the value of the encrypted attribute, set the value for the corresponding non-encrypted attribute.

   WebLogic Server converts the `String` to an encrypted byte array and sets it as `CustomIdentityKeyStorePassPhraseEncrypted`. (It does not set a value for `CustomIdentityKeyStorePassPhrase`).

   For example, to set the `CustomIdentityKeyStorePassPhraseEncrypted` from a remote JMX client, invoke the `MBeanServerConnection.setAttribute` for an attribute named `CustomIdentityKeyStorePassPhrase`.

For example:

```
public void editDefaultIIOPPassword(ObjectName cfgRoot, String password)
    throws Exception {
    // Get the ServerMBean from the DomainMBean
    ObjectName server = (ObjectName) connection.invoke(cfgRoot,
       "lookupServer",
        new Object[]{"myserver"},new String[]{"java.lang.String"});
    Attribute newpassword = new Attribute("DefaultIIOPPassword",
       "mypassword");
    connection.setAttribute(server, newpassword);
}
```

Back Up an Encrypted Value

To make a backup copy of a password, use the getter method of the MBean's encrypted value to retrieve the encrypted byte array. Then write the value of the byte array to a file. WebLogic Server does not provide APIs or other utilities for decrypting values that it has encrypted.

If you need to restore the password value, you can load the saved value into a byte array and pass it as a parameter to the `MBeanServerConnection.setAttribute` method (see "Set the Value of an Encrypted Attribute (Recommended Technique)" on page 5-21).

Note: Because each WebLogic Sever domain uses its own encryption algorithm, you must back up and restore passwords separately for each domain even if the unencrypted value for the password is the same for all domains.

Instead of backing up the same encrypted password for each domain, you can use the getter method of an MBean's corresponding **unencrypted** value. This getter unencrypts

the password and copies into a `String`. The `String` will not be erased from memory until it is garbage collected and the memory is reallocated.

CHAPTER 6

Managing Security Realms with JMX

A security realm comprises mechanisms for protecting WebLogic resources. Each security realm consists of a set of configured security providers, which are modular components that handle specific aspects of security. You can create a JMX client that uses the providers in a realm to add or remove security data such as users and groups. You can also create a client that adds or removes providers and makes other changes to the realm configuration.

The following sections describe managing security realms with JMX:

For more information about WebLogic Security, see *Understanding WebLogic Security*.

Understanding the Hierarchy of Security MBeans

Like other subsystems, the WebLogic Server security framework organizes its MBeans in a hierarchy that JMX clients can navigate without constructing JMX object names. However, the set of MBean types that are available in a security realm depends on which security providers you have installed in the realm, and the set of services that each security provider enables depends on how the provider was created.

The root of the security realm hierarchy is the `RealmMBean`. It contains all of the providers that have been configured for the realm. For example, its `Authorizers` attribute contains all

authorization providers that have been configured for the realm. WebLogic Server installs a default set of security providers; therefore, by default the `RealmMBean Authorizers` attribute contains a `DefaultAuthorizerMBean`. However, you can uninstall these default providers and replace them with any number of your own providers or third-party providers. For information about the default security providers, see Configuring WebLogic Security Providers and Configuring Authorization Providers in *Securing WebLogic Server*.

Base Provider Types and Mix-In Interfaces

Each security provider must extend a base provider type. For example, `DefaultAuthorizerMBean` extends `AuthorizerMBean`, and any custom or third-party authorization provider also extends `AuthorizerMBean`. If a JMX client gets the value of the `RealmMBean Authorizers` attribute, the MBean server returns all MBeans in the realm that extend `AuthorizerMBean`. The JMX client can iterate through the list of providers and select one based on the value of its `Name` attribute or other criteria.

Base provider types can be enhanced by extending a set of optional mix-in interfaces. For example, if an authentication provider extends the `UserEditorMBean`, then the provider can add users to the realm.

Security MBeans

WebLogic Server's Security MBeans configure security providers in a security realm. The following tables describe the MBeans that configure different types of security providers.

- Table 6-1. describes the MBeans that configure Authentication security providers, as well as the abstract MBean classes that Authentication providers must extend. In addition to the MBeans in this table, WebLogic Server includes configuration MBeans for each out-of-the-box Authentication provider.
- Table 6-2 describes the MBeans that configure security providers, other than Authentication security providers.
- Table 6-3 describes optional MBean mixin interfaces that security providers can support for management and utility purposes.

For more information about configuring WebLogic security providers, see Configuring WebLogic Security Providers and Configuring Authentication Providers in *Securing WebLogic Server*. Figure 6-1 illustrates where the MBeans are located in the configuration MBean hierarchy.

Table 6-1 MBeans for Authentication Security Providers

This MBean...	Configures...
`AuthenticationProviderMBean`	The base MBean for all MBean implementations that manage Authentication providers. If your Authentication provider uses the WebLogic Security SSPI to provide login services, then your MBean must extend `weblogic.management.security.authentication.Authenticator`. If your Authentication provider uses the WebLogic Security SPI to provide identity-assertion services, then your MBean must extend `weblogic.management.security.authentication.IdentityAsserter`. See `AuthenticationProviderMBean` in the *WebLogic Server MBean Reference*.
`AuthenticatorMBean`	The SSPI MBean that all Authentication providers with login services must extend. This MBean provides a `ControlFlag` to determine whether the Authentication provider is a REQUIRED, REQUISITE, SUFFICENT, or OPTIONAL part of the login sequence. See `AuthenticatorMBean` in the *WebLogic Server MBean Reference*.
`IdentityAsserterMBean`	The SSPI MBean that all Identity Assertion providers must extend. This MBean enables an Identity Assertion provider to specify the token types for which it is capable of asserting identity. See `IdentityAsserterMBean` in the *WebLogic Server MBean Reference*.
`ServletAuthenticationFilterMBean`	The SSPI MBean that all Servlet Authentication Filter providers must extend. This MBean is just a marker interface. It has no methods on it. See `ServletAuthenticationFilterMBean` in the *WebLogic Server MBean Reference*.

.

Table 6-2 MBeans for Other Security Providers

This MBean...	Configures...
`AdjudicatorMBean`	The SSPI MBean that all Adjudication providers must extend. See `AdjudicatorMBean` in the *WebLogic Server MBean Reference*.
`DefaultAdjudicatorMBean`	Configuration attributes for the WebLogic Adjudication provider. See `DefaultAdjudicatorMBean` in the *WebLogic Server MBean Reference*.
`AuditorMBean`	The SSPI MBean that all Auditing providers must extend. See `AuditorMBean` in the *WebLogic Server MBean Reference*.
`DefaultAuditorMBean`	Configuration attributes for the WebLogic Auditing provider. See `DefaultAuditorMBean` in the *WebLogic Server MBean Reference*.
`AuthorizerMBean`	The SSPI MBean that all Authorization providers must extend. See `AuthorizerMBean` in the *WebLogic Server MBean Reference*.
`DeployableAuthorizerMBean`	The SSPI MBean that must be extended by all Authorization providers that can store policies created while deploying a Web application or EJB. See `DeployableAuthorizerMBean` in the *WebLogic Server MBean Reference*.
`DefaultAuthorizerMBean`	Configuration attributes for the WebLogic Authorization provider. See `DefaultAuthorizerMBean` in the *WebLogic Server MBean Reference*.
`CredentialMapperMBean`	The SSPI MBean that all Credential Mapping providers must extend. See `CredentialMapperMBean` in the *WebLogic Server MBean Reference*.

Table 6-2 MBeans for Other Security Providers

This MBean...	Configures...
`DeployableCredentialMapperMBean`	The SSPI MBean that must be extended by all Credential Mapper providers that can store credential maps created while deploying a component. See `DeployableCredentialMapperMBean` in the *WebLogic Server MBean Reference*.
`DefaultCredentialMapperMBean`	Configuration attributes for the WebLogic Credential Mapping provider, a username/password Credential Mapping provider. See `DefaultCredentialMapperMBean` in the *WebLogic Server MBean Reference*.
`PKICredentialMapperMBean`	Configuration attributes for the PKI Credential Mapping provider, a key pair Credential Mapping provider. See `PKICredentialMapperMBean` in the *WebLogic Server MBean Reference*.
`SAMLCredentialMapperMBean`	Configuration attributes for the SAML Credential Mapping provider, a Security Assertion Markup Language Credential Mapping provider. See `SAMLCredentialMapperMBean` in the *WebLogic Server MBean Reference*.
`CertPathProviderMBean`	The base MBean for all certification path providers. See `CertPathProviderMBean` in the *WebLogic Server MBean Reference*.
`CertPathBuilderMBean`	The SSPI MBean that all certification path providers with CertPathBuilder services must extend. See `CertPathBuilderMBean` in the *WebLogic Server MBean Reference*.
`CertPathValidatorMBean`	The SSPI MBean that all certification path providers with CertPathValidator services must extend. See `CertPathValidatorMBean` in the *WebLogic Server MBean Reference*.

Table 6-2 MBeans for Other Security Providers

This MBean...	Configures...
`CertificateRegistryMBean`	Configures and manages the certificate registry. It is both a builder and a validator. It supports building from the end certificate, the end certificate's subject DN, the end certificate's issuer DN and serial number, and the end certificate's subject key identifier. See `CertificateRegistryMBean` in the *WebLogic Server MBean Reference*.
`WebLogicCertPathProviderMBean`	The SSPI MBean that all certification path providers with CertPathBuilder services must extend. See `WebLogicCertPathProviderMBean` in the *WebLogic Server MBean Reference*.
`RoleMapperMBean`	The base MBean for Role Mapping providers. A Role Mapping provider for a non-deployable module must extend this MBean directly. A Role Mapping provider for a deployable module must extend the `DeployableRoleMapperMBean`. See `RoleMapperMBean` in the *WebLogic Server MBean Reference*.
`DeployableRoleMapperMBean`	The SSPI MBean that must be extended by Role Mapping providers that can store roles created while deploying a Web application or EJB. See `DeployableRoleMapperMBean` in the *WebLogic Server MBean Reference*.
`DefaultRoleMapperMBean`	Configuration attributes for the WebLogic Role Mapping provider. See `DefaultRoleMapperMBean` in the *WebLogic Server MBean Reference*.

.

Table 6-3 MBean Mixin Interfaces for Security Providers

This MBean...	Configures...
`ContextHandlerMBean`	Provides a set of attributes for ContextHandler support. An Auditor provider MBean can optionally implement this MBean. See `ContextHandlerMBean` in the *WebLogic Server MBean Reference*.
`GroupEditorMBean`	Provides a set of methods for creating, editing, and removing groups. An Authentication provider MBean can optionally implement this MBean. See `GroupEditorMBean` in the *WebLogic Server MBean Reference*.
`GroupMemberListerMBean`	Provides a method for listing a group's members. An Authentication provider MBean can optionally implement this MBean. See `GroupMemberListerMBean` in the *WebLogic Server MBean Reference*.
`GroupMembershipHierarchyCacheMBean`	Provides configuration attributes that are required to support the Group Membership Hierarchy Cache. An Authentication provider MBean can optionally implement this MBean. See `GroupMembershipHierarchyCacheMBean` in the *WebLogic Server MBean Reference*.
`GroupReaderMBean`	Provides a set of methods for reading data about groups. An Authentication provider MBean can optionally implement this MBean. See `GroupReaderMBean` in the *WebLogic Server MBean Reference*.
`MemberGroupListerMBean`	Provides a method for listing the groups that contain a member. An Authentication provider MBean can optionally implement this MBean. See `MemberGroupListerMBean` in the *WebLogic Server MBean Reference*.

Table 6-3 MBean Mixin Interfaces for Security Providers

This MBean...	Configures...
`UserEditorMBean`	Provides a set of methods for creating, editing, and removing users. An Authentication provider MBean can optionally implement this MBean. See `UserEditorMBean` in the *WebLogic Server MBean Reference*.
`UserLockoutManagerMBean`	Lists and manages lockouts on user accounts. An Authentication provider MBean can optionally implement this MBean. See `UserLockoutManagerMBean` in the *WebLogic Server MBean Reference*.
`UserPasswordEditorMBean`	Provides two methods for changing a user's password. An Authentication provider MBean can optionally implement this MBean. See `UserPasswordEditorMBean` in the *WebLogic Server MBean Reference*.
`UserReaderMBean`	Provides a set of methods for reading data about users. An Authentication provider MBean can optionally implement this MBean. See `UserReaderMBean` in the *WebLogic Server MBean Reference*.
`UserRemoverMBean`	Provides a method for removing users. An Authentication provider MBean can optionally implement this MBean. See `UserRemoverMBean` in the *WebLogic Server MBean Reference*.
`RoleEditorMBean`	Provides a set of methods for creating, editing, and removing roles. A Role Mapping provider MBean can optionally implement this MBean. See `RoleEditorMBean` in the *WebLogic Server MBean Reference*.

Table 6-3 MBean Mixin Interfaces for Security Providers

This MBean...	Configures...
`RoleListerMBean`	Provides a set of methods for listing data about roles. A Role Mapping provider MBean can optionally implement this MBean. See `RoleListerMBean` in the *WebLogic Server MBean Reference*.
`RoleReaderMBean`	Provides a set of methods for reading roles. A Role Mapping provider MBean can optionally implement this MBean. See `RoleReaderMBean` in the *WebLogic Server MBean Reference*.
`PolicyEditorMBean`	Provides a set of methods for creating, editing, and removing policies. An Authorization provider MBean can optionally implement this MBean. See `PolicyEditorMBean` in the *WebLogic Server MBean Reference*.
`PolicyListerMBean`	Provides a set of methods for listing data about policies. An Authorization provider MBean can optionally implement this MBean. See `PolicyListerMBean` in the *WebLogic Server MBean Reference*.
`PolicyReaderMBean`	Provides a set of methods for reading policies. An Authorization provider MBean can optionally implement this MBean. See `PolicyReaderMBean` in the *WebLogic Server MBean Reference*.
`PKICredentialMapEditorMBean`	Provides a set of methods for creating, editing, and removing a credential map that matches users, resources and credential action to keystore aliases and the corresponding passwords. A PKICredentialMapping provider MBean can optionally implement this MBean. See `PKICredentialMapEditorMBean` in the *WebLogic Server MBean Reference*.

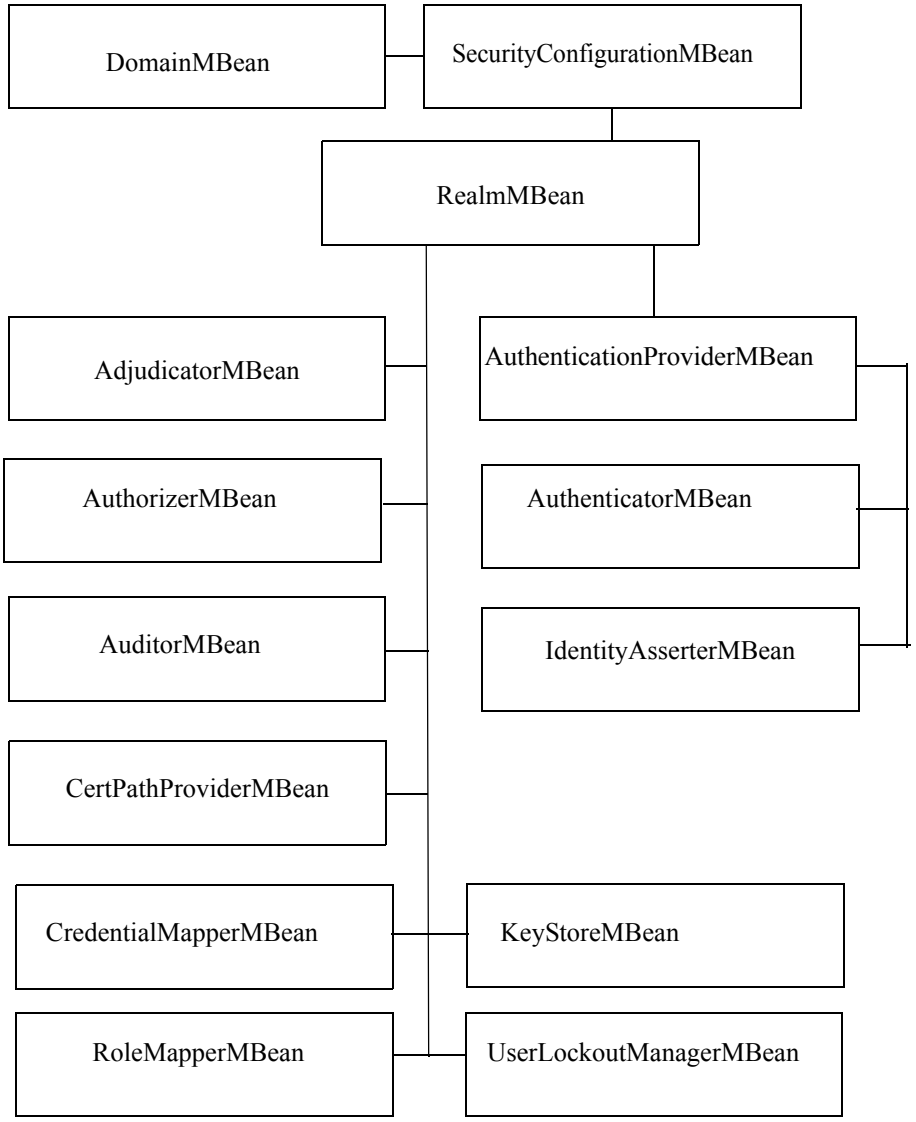
Table 6-3 MBean Mixin Interfaces for Security Providers

This MBean...	Configures...
`PKICredentialMapReaderMBean`	Provides a set of methods for reading a credential map that matches users and resources to keystore aliases and their corresponding passwords that can then be used to retrieve key information or public certificate information from the configured keystores. A PKICredentialMapping provider MBean can optionally implement this MBean. See `PKICredentialMapReaderMBean` in the *WebLogic Server MBean Reference*.
`UserPasswordCredentialMapEditorMBean`	Provides a set of methods for creating, editing, and removing a credential map that matches WebLogic users to remote usernames and their corresponding passwords. A Credential Mapping provider MBean can optionally extend this MBean. See `UserPasswordCredentialMapEditorMBean` in the *WebLogic Server MBean Reference*.
`UserPasswordCredentialMapExtendedReaderMBean`	Provides a set of methods for reading credentials and credential mappings. Credential mappings match WebLogic users to remote usernames and passwords. A Credential Mapping provider MBean can optionally extend this MBean. See `UserPasswordCredentialMapExtendedReaderMBean` in the *WebLogic Server MBean Reference*.
`UserPasswordCredentialMapReaderMBean`	Provides a set of methods for reading credentials and credential mappings. Credential mappings match WebLogic users to remote usernames and passwords. A Credential Mapping provider MBean can optionally extend this MBean. See `UserPasswordCredentialMapReaderMBean` in the *WebLogic Server MBean Reference*.
`ImportMBean`	Provides a set of methods for importing provider specific data. An optional mixin interface that any security provider may extend. See `ImportMBean` in the *WebLogic Server MBean Reference*.

Table 6-3 MBean Mixin Interfaces for Security Providers

This MBean...	Configures...
`ExportMBean`	Provides a set of methods for exporting provider specific data. An optional mixin interface that any security provider may extend. See `ExportMBean` in the *WebLogic Server MBean Reference.*
`ListerMBean`	Provides a general mechanism for returning lists. Derived MBeans extend this interface to add methods that access the data of the current object in the list. An optional mixin interface that any security provider may extend. See `ListerMBean` in the *WebLogic Server MBean Reference.*
`NameListerMBean`	Defines a method used to return lists of names. An optional mixin interface that any security provider may extend. See `NameListerMBean` in the *WebLogic Server MBean Reference.*
`LDAPServerMBean`	Provides methods to get configuration parameters needed for connecting to an external LDAP server. An optional mixin interface that any security provider may extend. See `LDAPServerMBean` in the *WebLogic Server MBean Reference.*
`ApplicationVersionerMBean`	The SSPI MBean that security providers extend to indicate that the provider supports versionable applications. An optional mixin interface that a RoleMapper, Authorizer, or CredentialMapper provider MBean may extend. See `ApplicationerVersionMBean` in the *WebLogic Server MBean Reference.*

Figure 6-1 Security MBeans



Choosing an MBean Server to Manage Security Realms

When using JMX to manage security realms, you must use two different MBean servers depending on your task:

- To set the value of a security MBean attribute, you must use the Edit MBean Server.
- To add users, groups, roles, and policies, or to invoke other operations in a security provider MBean, you must use a Runtime MBean Server or the Domain Runtime MBean Server.

  In addition, to prevent the possibility of incompatible changes, you cannot invoke operations in security provider MBeans if your client or another JMX client has an edit session currently active.

For example, the value of the `MinimumPasswordLength` attribute in `DefaultAuthenticatorMBean` is stored in the domain's configuration document. Because all modifications to this document are controlled by WebLogic Server, to change the value of this attribute you must use the Edit MBean Server and acquire a lock on the domain's configuration. The `createUser` operation in `DefaultAuthenticatorMBean` adds data to an LDAP server, which is not controlled by WebLogic Server. To prevent incompatible changes between the `DefaultAuthenticatorMBean`'s configuration and the data that it uses in the LDAP server, you cannot invoke the `createUser` operation if you or other users are in the process of modifying the `MinimumPasswordLength` attribute. In addition, because changing this attribute requires you to restart WebLogic Server, you cannot invoke the `createUser` operation until you have restarted the server.

Working with Existing Security Providers

Because security providers can extend optional mix-in interfaces, not all security providers can perform all tasks. This flexibility enables your organization's security architect to design a realm for your security needs. The flexibility also makes the design of your JMX clients dependent upon the design and configuration of each realm.

For example, some realms might contain three types of Authentication providers:

- One that extends `UserEditorMBean` to save administrative users to an LDAP server
- One that extends `UserEditorMBean` to save customers to a database management system
- One that does not extend `UserEditorMBean` and is used only to authenticate existing users

To work with the Authentication providers in this realm, your JMX client must be able to determine which one can add users to the appropriate repository.

Table 6-4 discusses techniques for finding a security provider that is appropriate for your task.

Table 6-4 Finding a Provider in the Realm

Technique	Description
Find by name	Each security provider instance is assigned a short name when an administrator configures it for the realm. Your JMX client can look up all providers of a specific type (such as all Authentication providers) and choose the one that matches a name. For an example of such a JMX client, start the WebLogic Server Examples Server. From the Examples Server home page, click on "Extending a Realm Using JMX." The source for this JMX client is installed as *WL_HOME*/samples/server/medrec/src/medrecEar/adminWebApp/WEB-INF/src/com/bea/medrec/actions/CreateNewAdminAction.java where *WL_HOME* is the location in which you installed WebLogic Server. If you use this technique, consider saving the name of the security provider in a configuration file instead of hard-coding it in your JMX client. The configuration file enables system administrators to change the providers in the realm and update the properties file instead of requiring you to update and recompile the JMX client.

Table 6-4 Finding a Provider in the Realm

Technique	Description
Find by MBean type	If the system administrator always wants to use the same type of provider for a task, then your JMX client can find the provider MBean that is of the specified type. For example, if the system administrator always wants to use a `SQLAuthenticatorMBean` to add customers to a realm, your JMX client can find an instance of `SQLAuthenticatorMBean`. While this technique requires no user input, it assumes: • There will always be an instance of `SQLAuthenticatorMBean` in the realm and this one instance extends `UserEditorMBean`. • If there are multiple instances of `SQLAuthenticatorMBean`, all of them extend `UserEditorMBean` and it does not matter which instance is used. See "Discovering Available Services" on page 6-15.
Use any provider that extends the mix-in interface you need	You can create a JMX client that learns about the class hierarchy for each provider MBean instance and chooses an instance that extends the mix-in interface you need for your task. For example, your client can discover which Authentication provider extends `UserEditorMBean`. See "Discovering Available Services" on page 6-15. Use this technique if you know that your security realm will contain only one MBean that extends the needed mix-in interface, or if it does not matter which one you use.

Discovering Available Services

To create a JMX client that finds MBeans by type or mix-in interface:

1. Connect to a WebLogic Server Runtime MBean Server. See "Make Remote Connections to an MBean Server" on page 4-2.

   All WebLogic Server instances maintain their own Runtime MBean Server, and you can connect to any server's Runtime MBean Server.

2. Get all security provider MBeans of a specific type in the realm (for example, get all Authentication provider MBeans):

a. Use either the `RuntimeServiceMBean` or `DomainRuntimeServiceMBean` to navigate the following path through the WebLogic Server MBean hierarchy: `DomainMBean` to `SecurityConfigurationMBean` to `RealmMBean`.

   See "Navigate MBean Hierarchies" on page 4-8.

b. Get the value of the `RealmMBean` attribute that contains instances of the security provider type.

   For example, to get all Authentication providers, get the value of the `RealmMBean` `AuthenticationProviders` attribute.

3. For each security provider MBean in the `RealmMBean` attribute, get the name of the MBean's class (see Listing 6-1):

   a. Get the provider MBean's `javax.management.ModelMBeanInfo` object.

      Use `MBeanServerConnection.getMBeanInfo(`*`Provider-MBean`*`)` where *`Provider-MBean`* is a provider MBean that you retrieved from `RealmMBean`.

   b. Get the MBean info's `javax.management.Descriptor` object, and then get the value of the Descriptor's `interfaceClassName` field.

4. Use the WebLogic Server MBean type service to find all security provider MBean classes that extend a particular base type or mix-in interface (see Listing 6-1):

   a. Determine the fully-qualified interface name of the base type or mix-in interface.

      Each entry in the *WebLogic Server MBean Reference* lists the fully-qualified interface name of WebLogic Server provider MBeans. If you use a third-party provider, refer to the third-party documentation for this information.

      For example, the fully-qualified interface name of the `UserEditorMBean` mix-in interface is `weblogic.management.security.authentication.UserEditorMBean`. (See `UserEditorMBean` in the *WebLogic Server MBean Reference*.)

   b. Construct the `MBeanTypeService` MBean's object name.

      The `MBeanTypeService` MBean is always registered under the following `javax.management.ObjectName`:
      `com.bea:Name=MBeanTypeService,Type=weblogic.management.mbeanservers.MBeanTypeService`

   c. Invoke the `MBeanTypeService` MBean's `getSubtypes( java.lang.String` *`beanInterface`* `)` operation, where:

beanInterface is the fully-qualified interface name that you determined in Step 1.

The operation returns an array of `java.lang.String` objects.

5. Compare the output of the MBean type service with the class name of each provider MBean instance (see Listing 6-1).
6. If the provider MBean's class implements or extends the interface from step 4a, invoke operations on the provider MBean.

Listing 6-1 Example: Determine If a Provider MBean Instance Extends UserEditorMBean Mix-In Interface

```
ObjectName MBTservice = new ObjectName(
   "com.bea:Name=MBeanTypeService,Type=weblogic.management.mbeanservers.
   MBeanTypeService");

for (int p = 0; atnProviders != null && p < atnProviders.length; p++) {
   ModelMBeanInfo info = (ModelMBeanInfo)
   mBeanServerConnection.getMBeanInfo(atnProviders[p]);
   Descriptor desc = info.getMBeanDescriptor();
   String className = (String)desc.getFieldValue("interfaceClassName");
   String[] mba =  (String[]) mBeanServerConnection.invoke( MBTservice,
      "getSubtypes", new Object[] {
      "weblogic.management.security.authentication.UserEditorMBean" },
      new String[] { "java.lang.String" });
   boolean isEditor = false;
   for (int i = 0; i < mba.length; i++) {
      if (mba[i].equals(className)){
         userEditor = atnProviders[p];
         isEditor = true;
         break;
      }
      if (isEditor = true) break;
   }
}
```

Example: Adding Users to a Realm

The code example in Listing 6-2 adds a user to a security realm and adds the user to the `Administrators` group by searching through all of the authentication providers in the realm and using the first one that extends `UserEditorMBean`.

Note the following about the code example:

- Similar to the code in the MedRec example domain, the user name and password come from a JavaBean that was created from an Apache Struts action.

  To see the MedRec code:

  a. Start the WebLogic Server Examples Server.

  b. From the Examples Server home page, click on "Extending a Realm Using JMX."

- The code does not need to lock the domain's configuration because it is not modifying the configuration of the security MBean itself. Instead, it is invoking an operation in the default Authorization provider which saves security data in an LDAP server.

Listing 6-2 Example: Adding Users to a Realm

```
public ActionForward createNewAdmin(ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
   throws ClientException, Exception {
   logger.info("Create New Admin");
   CreateAdminBean user = (CreateAdminBean) form;
   logger.debug(user.toString());

   MBeanServerConnection mBeanServerConnection =
      this.getDomainMBeanServerConnection(request);
   ObjectName service = new
      ObjectName("com.bea:Name=DomainRuntimeService,"+
      "Type=weblogic.management.mbeanservers.domainruntime.
      DomainRuntimeServiceMBean");
   ObjectName domainMBean =
      (ObjectName) mBeanServerConnection.getAttribute(service,
      "DomainConfiguration");
   ObjectName securityConfiguration =
      (ObjectName) mBeanServerConnection.getAttribute(domainMBean,
      "SecurityConfiguration");
   ObjectName defaultRealm =
      (ObjectName) mBeanServerConnection.
      getAttribute(securityConfiguration, "DefaultRealm");
   ObjectName[] atnProviders =
      (ObjectName[]) mBeanServerConnection.getAttribute(defaultRealm,
      "AuthenticationProviders");

   ObjectName userEditor = null;
   ObjectName MBTservice = new ObjectName(
      "com.bea:Name=MBeanTypeService,Type=weblogic.management.mbeanservers.
      MBeanTypeService");
```

```
for (int p = 0; atnProviders != null && p < atnProviders.length; p++) {
   ModelMBeanInfo info = (ModelMBeanInfo)
   mBeanServerConnection.getMBeanInfo(atnProviders[p]);
   Descriptor desc = info.getMBeanDescriptor();
   String className = (String)desc.getFieldValue("interfaceClassName");
   String[] mba =  (String[]) mBeanServerConnection.invoke( MBTservice,
      "getSubtypes", new Object[] {
      "weblogic.management.security.authentication.UserEditorMBean" },
      new String[] { "java.lang.String" });
   boolean isEditor = false;
   for (int i = 0; i < mba.length; i++) {
      if (mba[i].equals(className)){
         userEditor = atnProviders[p];
         isEditor = true;
         break;
      }
      if (isEditor = true) break;
   }
}

try {
   mBeanServerConnection.invoke(
      userEditor, "createUser",
      new Object[] {user.getUsername(), user.getPassword(),
         "MedRec Admininistator"},
      new String[] {"java.lang.String", "java.lang.String",
          "java.lang.String"}
   );
} catch (MBeanException ex) {
   Exception e = ex.getTargetException();
   if (e instanceof AlreadyExistsException) {
      logger.info("User, " + user.getUsername() + ", already exists.");
      ActionErrors errors = new ActionErrors();
      errors.add("invalidUserName",
         new ActionError("invalid.username.already.exists"));
      saveErrors(request, errors);
      return mapping.findForward("create.new.admin");
   } else {
      logger.debug(e);
      return mapping.findForward("create.new.admin");
   }
}

try {
   mBeanServerConnection.invoke(
      userEditor, "addMemberToGroup",
      new Object[] {"Administrators", user.getUsername()},
      new String [] {"java.lang.String", "java.lang.String"}
   );
```

```
        mBeanServerConnection.invoke(
          userEditor, "addMemberToGroup",
          new Object[] {"MedRecAdmins", user.getUsername()},
          new String [] {"java.lang.String", "java.lang.String"}
        );
    } catch (MBeanException ex) {
        Exception e = ex.getTargetException();
        if (e instanceof NameNotFoundException) {
          logger.info("Invalid Group Name.");
          ex.printStackTrace();
          return mapping.findForward("create.new.admin");
        } else {
          logger.debug(e);
          return mapping.findForward("create.new.admin");
        }
    }
    logger.info("MedRec Administrator successfully created.");
    return mapping.findForward("create.new.admin.successful");
}
```

Modifying the Realm Configuration

While security provider MBeans handle specific aspects of security, such as authentication and authorization, two other MBeans handle general, realm-wide and domain-wide aspects of security:

- `RealmMBean` represents a security realm. JMX clients can use it to add or remove security providers and to specify such behaviors as whether Web and EJB containers call the security framework on every access or only when security is set in the deployment descriptors.
- `SecurityConfigurationMBean` specifies domain-wide security settings such as connection filters and URL-pattern matching behavior for security constraints, servlets, filters, and virtual-hosts in the WebApp container and external security policies.

These two MBeans persist their data in WebLogic Server configuration files. Therefore, to modify attribute values in `RealmMBean` or `SecurityConfigurationMBean`, you must use the Edit MBean Server and `ConfigurationManagerMBean` as described in "Managing a Domain's Configuration with JMX" on page 5-1.

CHAPTER 7

Using Notifications and Monitor MBeans

JMX provides two ways to monitor MBeans: MBeans can emit notifications when specific events occur (such as a change in an attribute value), or a special type of MBean called a monitor MBean can poll another MBean and periodically emit notifications to describe an attribute value. You create Java classes called **listeners** that listen for these notifications and respond appropriately. For example, your management utility can include a listener that receives notifications when applications are deployed, undeployed, or redeployed.

All WebLogic Server configuration MBeans emit notifications when attribute values change, and some runtime MBeans do.

The following sections describe working with notifications and listeners:

Best Practices: Listening Directly Compared to Monitoring

If the MBean that you want to monitor emits notifications, you can choose whether to create a listener object that listens for changes in the MBean or a monitor MBean that periodically polls the MBean and emits notifications only when its attributes change in specific ways. The

technique 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 because the MBean pushes its notifications to your listener and you are notified of a change almost immediately. However, the base classes that you implement for a listener and optional filter (`javax.management.NotificationListener` and `NotificationFilter`) provide few facilities for comparing values with thresholds and other values. (See the `javax.management` package in the *J2SE 5.0 API Specification*.)

If your notification requirements are sufficiently complex, or if you want to monitor a group of changes that are not directly associated with a single change in the value of an MBean attribute, use a monitor MBean. (See the `javax.management.monitor` package in the *J2SE 5.0 API Specification*.) The monitor MBeans provide a rich set of tools for comparing data and sending notifications only under 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: Listening for WebLogic Server Events

The WebLogic Server JMX agent and WebLogic Server MBeans emit different types of notification objects for different types of events. Many event types trigger multiple MBeans to emit notifications at different points within the event process. Table 7-1 describes common event types and recommends the MBean with which a JMX monitoring application should register to listen for notifications.

Note: Each JMX notification object contains an attribute named `Type`, which contains a dot-delimited string. Do not confuse discussions of this `Type` attribute with a notification's object type.

The `Type` attribute offers a way to categorize and filter notifications. For example, if your custom MBeans emit notifications, JMX conventions suggest that you set your notification object's `Type` attribute to a string that starts with your company name: `mycompany.myapp.valueIncreased`.

All JMX notification objects extend the `javax.management.Notification` object type. JMX and WebLogic Server define additional notification object types, such as `javax.management.AttributeChangeNotification`. The additional object types contain specialized sets of information that are appropriate for different types of events. (See the list of `Notification` subclasses for `javax.management.Notification` in the *J2SE 5.0 API Specification*. Also see

`weblogic.management.logging.WebLogicLogNotification` in the *WebLogic Server API Reference*.)

Table 7-1 Events and Notification Objects

Event	Listening Recommendation
A WebLogic Server instance starts or stops	To receive a notification when a server starts or stops, register a listener with each server's `ServerLifeCycleRuntimeMBean` in the Domain Runtime MBean Server and configure an `AttributeChangeNotificationFilter`. Each server in a domain provides its own `ServerLifeCycleRuntimeMBean`, which is available through the Domain Runtime MBean Server even if the server itself is not active. When you start a server instance, the server's `ServerLifeCycleRuntimeMBean` updates the value of its `State` attribute and emits an `AttributeChangeNotification`. For an example of such a listener and filter, see "Listening for Notifications from WebLogic Server MBeans: Main Steps" on page 7-7. **Note:** This recommendation assumes that you start a domain's Administration Server before starting Managed Servers. If a Managed Server starts before the Administration Server, a listener in the Domain Runtime MBean Server (which runs only on the Administration Server) will not be initialized at the time the Managed Server's `ServerLifeCycleRuntimeMBean` changes its state to `RUNNING`. If you cannot guarantee that the Administration Server starts first, use the JMX timer service to periodically query the Domain Runtime MBean Server for MBeans whose object name contains the `Type=ServerRuntime` key property. An MBean that matches this query is a `ServerRuntimeMBean`, which each server instance creates as part of its startup process. If the query finds a newly created `ServerRuntimeMBean`, you know that a new server instance has been started. See `MBeanServerConnection queryNames`.

Table 7-1 Events and Notification Objects

A WebLogic Server resource is created or destroyed	When you create a resource such as a server or a JDBC data source, WebLogic Server registers the resource's configuration MBean in the MBean server. When you delete a resource, WebLogic Server unregisters the configuration MBean. To listen for the registration and unregistration of MBeans, register a listener with `javax.management.MBeanServerDelegate`, which emits notifications of type `javax.management.MBeanServerNotification` when MBeans are registered or unregistered. If you register a listener with `MBeanServerDelegate` in the Edit MBean Server, you receive notifications when someone modifies the pending MBean hierarchy. If you register a listener in the Runtime MBean Server or the Domain Runtime MBean Server, you receive notifications only when pending changes have been successfully activated in the domain. If you are interested solely in monitoring configuration data (and are not interested in monitoring runtime statistics), register your listener in only one Runtime MBean Server. See "Best Practices: Choosing an MBean Server" on page 4-5. See "Example: Listening for The Registration of Configuration MBeans" on page 7-16.
The configuration of a WebLogic Server resource is modified	All configuration MBeans emit notifications of type `AttributeChangeNotification` when their attribute values change. To receive this notification, register a listener with the MBean that is in the Domain Runtime MBean Server or Runtime MBean Server (see "Best Practices: Choosing an MBean Server" on page 4-5). If you register an MBean in the Edit MBean Server, you receive notifications when someone modifies the pending MBean hierarchy. If you register a listener in the Runtime MBean Server or the Domain Runtime MBean Server, you receive notifications only when pending changes have been successfully activated in the domain. If you are interested solely in monitoring configuration data (and are not interested in monitoring runtime statistics), register your listener in only one Runtime MBean Server. See "Best Practices: Choosing an MBean Server" on page 4-5.

Table 7-1 Events and Notification Objects

The runtime state of a WebLogic Server resource changes	Some runtime MBeans emit notifications of type `AttributeChangeNotification` when their attribute values change. To receive this notification, register a listener with the MBean in the Domain Runtime MBean Server. If a runtime MBean does not emit notifications, you can create a monitor MBean that polls the runtime MBean. See "Using Monitor MBeans to Observe Changes: Main Steps" on page 7-21.
A WebLogic Server resource emits a log message	When a WebLogic Server resource generates a log message, the server's `weblogic.management.runtime.LogBroadcasterRuntimeMBean` emits a notification of type `weblogic.management.logging.WebLogicLogNotification`, which can be cast as the standard `javax.management.Notification` class. To listen for log message notifications, register a listener with `LogBroadcasterRuntimeMBean`. You can listen for the standard JMX notifications, or if you want to retrieve detailed information about the log messages, listen for `WebLogicLogNotifications`, which contains methods that you can use to retrieve detailed information. Listening for `WebLogicLogNotifications` requires you to import this WebLogic Server class into your listener class. To see a list of error messages that WebLogic Server resources generate, refer to *WebLogic Server Message Catalogs*. For more information, see `WebLogicLogNotification` in the *WebLogic Server API Reference*.

Best Practices: Listening or Monitoring WebLogic Server Runtime Statistics

WebLogic Server MBeans provide detailed statistics on the runtime state of its services and resources. The statistics in Table 7-2 provide a general overview of the performance of WebLogic Server. You can listen for changes to these statistics by creating a listener and registering it directly with the MBeans that contain the attributes or you can configure monitor MBeans to periodically poll and report only the statistics that you consider to be significant. (See "Registering a Notification Listener and Filter" on page 7-11 and "Registering the Monitor and Listener" on page 7-24.)

Table 7-2 Commonly Monitored WebLogic Server Runtime Statistics

To track this statistic...	Listen or monitor this MBean attribute...
The current state of server.	MBean Type: `ServerLifeCycleRuntimeMBean` Attribute Name: `State`
Activity on the server's listen ports.	MBean Type: `ServerRuntimeMBean` Attribute Name: `OpenSocketsCurrentCount` MBean Type: `ServerMBean` Attribute Name: `AcceptBacklog` 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.
Memory and thread use.	MBean Type: `ThreadPoolRuntimeMBean` Attribute Name: `ExecuteThreadIdleCount` Indicates the number of threads in a server's execute queue that are taking up memory space but are not being used to process data.
	MBean Type: `ThreadPoolRuntimeMBean` Attribute Name: `PendingUserRequestCount` Indicates the number of user requests waiting in a server's execute queue.
	MBean Type: `JVMRuntimeMBean` Attribute Name: `HeapSizeCurrent` Indicates the amount of memory (in bytes) that is currently available in the server's JVM heap.

Table 7-2 Commonly Monitored WebLogic Server Runtime Statistics

To track this statistic...	Listen or monitor this MBean attribute...
Database connections.	MBean Type: `JDBCDataSourceRuntimeMBean` Attribute Name: `ActiveConnectionsCurrentCount` Indicates the current number of active connections in a JDBC connection pool.
	MBean Type: `JDBCDataSourceRuntimeMBean` Attribute Name: `ActiveConnectionsHighCount` 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: `JDBCDataSourceRuntimeMBean` Attribute Name: `LeakedConnectionCount` Indicates the total number of leaked connections. 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.
	MBean Type: `JDBCDataSourceRuntimeMBean` Attribute Name: `ConnectionDelayTime` Indicates the average time to connect to a connection pool.
	MBean Type: `JDBCDataSourceRuntimeMBean` Attribute Name: `FailuresToReconnectCount` Indicates when the connection pool fails to reconnect to its data store. 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 directly for the notifications that an MBean emits:

1. Create a listener class in your application. See "Creating a Notification Listener" on page 7-8.

2. Create an additional class that registers your listener and an optional filter with the MBean whose notifications you want to receive. See "Configuring a Notification Filter" on page 7-10 and "Registering a Notification Listener and Filter" on page 7-11.
3. Package and deploy the listener and registration class. See "Packaging and Deploying Listeners on WebLogic Server" on page 7-14.

Creating a Notification Listener

To create a notification listener:

1. Create a class that implements `javax.management.NotificationListener`.

   See `NotificationListener` in the *J2SE 5.0 API Specification*.

2. Within the class, add a `NotificationListener.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. (Optional) In most listening situations, you want to know more than the simple fact that an MBean has emitted a notification object. For example, you might want to know the value of the notification object's `Type` attribute, which is used to classify the type of event that caused the notification to be emitted.

   To retrieve information from a notification object, within your `handleNotification` method invoke the object's methods. Because all notification types extend `javax.management.Notification`, the following `Notification` methods are available for all notifications:

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

   See `Notification` in the *J2SE 5.0 API Specification*.

   Most notification types provide additional methods for retrieving data that is specific to the notification. For example, `javax.management.AttributeChangeNotification` provides `getNewValue()` and `getOldValue()`, which you can use to determine how the attribute value has changed.

Listing 7-1 is a simple listener that uses `AttributeChangeNotification` methods to retrieve the name of an attribute with a changed value, and the old and new values.

Listing 7-1 Notification Listener

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

public class MyListener implements NotificationListener {

    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() );
        }
    }
}
```

Listening from a Remote JVM

As of JMX 1.2, there are no special requirements for programming a listener that runs in a different JVM from the MBean to which it is listening.

Once you establish a connection to the remote JMX agent (using `javax.management.MBeanServerConnection`), JMX takes care of sharing data between the JVMs. See "Registering a Notification Listener and Filter" on page 7-11 for instructions on establishing a connection from a remote JVM.

Best Practices: Creating a Notification Listener

Consider the following recommendations while creating your `NotificationListener` class:

- Unless you 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 data sources, 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 `JDBCDataSourceRuntimeMBean`.

- 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.

- Your implementation of the `handleNotification` method should return as soon as possible to avoid blocking its notification broadcaster.

- If you invoke a method from a specialized notification type, wrap the method calls in an `if` statement to prevent your listener from invoking the method on notification objects of all types.

Configuring a Notification Filter

As of JDK 1.5, the JDK includes two simple filter classes that you can configure to forward notifications that match criteria that you specify. To configure one of the JDK's filter classes:

1. In the class that registers your listener with an MBean create an instance of `javax.management.NotificationFilterSupport` or `AttributeChangeNotificationFilter`.

2. Invoke a filter class method to specify filter criteria.

   See `NotificationFilterSupport` or `AttributeChangeNotificationFilter` in the *J2SE 5.0 API Specification*.

For example, the following lines of code configure an `AttributeChangeNotificationFilter` that forwards only attribute change notifications and only if there is a change in an attribute named `State`:

```
AttributeChangeNotificationFilter filter =
   new AttributeChangeNotificationFilter();
filter.enableAttribute("State");
```

Creating a Custom Filter

If the JDK's filter class is too simplistic for your needs, you can create more sophisticated, custom filter classes. (See `NotificationFilter` in the *J2SE 5.0 API Specification*.) However, BEA recommends that you use the JDK filter classes whenever possible: using a custom filter complicates the packaging and deployment of your listener and filter. See "Packaging and Deploying Listeners on WebLogic Server" on page 7-14.

Registering a Notification Listener and Filter

After you implement a notification listener class, you create an additional class that registers your listener (and optionally configures and registers a filter) with an MBean instance.

To register a notification listener and filter with an MBean:

1. Initialize a connection to the Domain Runtime MBean Server.

   See "Make Remote Connections to an MBean Server" on page 4-2.

2. To register with a WebLogic Server MBean, navigate the MBean hierarchy and retrieve an object name for the MBean that you want to listen to. See "Navigate MBean Hierarchies" on page 4-8.

   To register with a custom MBean, create an `ObjectName` that contains the MBean's JMX object name. See `javax.management.ObjectName` in the *J2SE 5.0 API Specification*.

3. Instantiate the listener class that you created.

4. (Optional) Instantiate and configure one of the JDK's filter classes or instantiate a custom class.

5. Register the listener and filter by passing the MBean's object name, listener class, and filter class to the `MBeanServerConnection.addNotificationListener (ObjectName` *`name`*`, ObjectName` *`listener`*`, NotificationFilter` *`filter`*`, Object` *`handback`*`)` method.

The example class registers the listener from Listing 7-1 and the JDK's `AttributeChangeNotificationFilter` with all `ServerLifeCycleRuntimeMBean`s in a domain. The class does not pass a handback object.

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 that keeps the `RegisterListener` class active and not exit the main program. 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 7-2 Registering a Listener with ServerLifeCycleRuntimeMBean

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

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

import javax.management.AttributeChangeNotificationFilter;

public class RegisterListener {
   private static MBeanServerConnection connection;
   private static JMXConnector connector;
   private static final ObjectName service;
   // Initializing the object name for DomainRuntimeServiceMBean
   // so it can be used throughout the class.
   static {
      try {
         service = new ObjectName(
         "com.bea:Name=DomainRuntimeService,Type=weblogic.management.mbeanserv
          ers.domainruntime.DomainRuntimeServiceMBean");
      }catch (MalformedObjectNameException e) {
         throw new AssertionError(e.getMessage());
      }
   }

   /*
    * Initialize connection to the Domain Runtime MBean server
    * each server in the domain hosts its own instance.
    */
   public static void initConnection(String hostname, String portString,
```

```
   String username, String password) throws IOException,
   MalformedURLException {
   String protocol = "t3";
   Integer portInteger = Integer.valueOf(portString);
   int port = portInteger.intValue();
   String jndiroot = "/jndi/";
   String mserver = "weblogic.management.mbeanservers.domainruntime";
   JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname, port,
      jndiroot + mserver);

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

/*
 * Get an array of ServerLifeCycleRuntimeMBeans
 */
public static ObjectName[] getServerLCRuntimes() throws Exception {
   ObjectName domainRT = (ObjectName) connection.getAttribute(service,
      "DomainRuntime");
   return (ObjectName[]) connection.getAttribute(domainRT,
      "ServerLifecycleRuntimes");
}

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

   try {
      //Instantiating your listener class.
      MyListener listener = new MyListener();
      AttributeChangeNotificationFilter filter =
         new AttributeChangeNotificationFilter();
      filter.enableAttribute("State");

      initConnection(hostname, portString, username, password);

      //Passing the name of the MBeans and your listener class to the
      //addNotificationListener method of MBeanServer.
      ObjectName[] serverLCRT = getServerLCRuntimes();
      int length= (int) serverLCRT.length;
      for (int i=0; i < length; i++) {
         connection.addNotificationListener(serverLCRT[i], listener,
```

```
                filter, null);
             System.out.println("\n[myListener]: Listener registered with"
                +serverLCRT[i]);
          }

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

Packaging and Deploying Listeners on WebLogic Server

You can package and deploy a JMX listener as a remote application, a WebLogic Server startup class (which makes the listener available as soon as a server boots), or within one of your other applications that you deploy on WebLogic Server.

If you use a filter from the JDK, you do not need to package the filter class. It is always available through the JDK.

Table 7-3 describes how to package and deploy your listeners and any custom filters.

Table 7-3 Packaging and Deploying Listeners and Custom Filters

If you deploy the listener...	Do this for the listener...	Do this for a custom filter...
As a remote application	Make the listener's class available on the remote client's classpath.	Make the filter's class available on the remote client's classpath. Also add the filter class to the classpath of each server instance that hosts the monitored MBeans by archiving the class in a JAR file and copying the JAR in each server's `lib` directory. See Domain Directory Contents in *Understanding Domain Configuration*.

Table 7-3 Packaging and Deploying Listeners and Custom Filters

If you deploy the listener...	Do this for the listener...	Do this for a custom filter...
As a WebLogic Server startup class	Add the listener class to the server's classpath by archiving the class in a JAR file and copying the JAR in the server's `lib` directory.	Add the filter class to the server's classpath by archiving the class in a JAR file and copying the JAR in the server's `lib` directory. See Domain Directory Contents in *Understanding Domain Configuration*.
As part of an application that you deploy on WebLogic Server	Package the listener class with the application.	Package the listener class with the application. Also add the filter class to the classpath of each server instance that hosts the monitored MBeans by doing one of the following: • Archiving the class in a JAR file and copying the JAR in each server's `lib` directory. See Domain Directory Contents in *Understanding Domain Configuration*. • Using the JMX MLet service to make the filter class available to the MBean server. See `javax.management.loading.MLet` in the *J2SE 5.0 API Specification* and the JMX 1.2 specification, which you can download from http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html.

Example: Listening for The Registration of Configuration MBeans

When you create a WebLogic Server resource, such as a server or a JDBC data source, WebLogic Server creates a configuration MBean and registers it in the Domain Runtime MBean Server.

To listen for these events, register a listener with `javax.management.MBeanServerDelegate`, which emits a notification of type `javax.management.MBeanServerNotification` each time an MBean is registered or unregistered. See `MBeanServerDelegate` in the *J2SE 5.0 API Specification*.

Note the following about the example listener in Listing 7-3:

- To provide information about which type of WebLogic Server MBean has been registered, the listener looks at the object name of the registered MBean. All WebLogic Server MBean object names contain a key property whose name is "Type" and whose value indicates the type of MBean. For example, instances of `ServerRuntimeMBean` contain the `Type=ServerRuntime` key property in their object names.
- All JMX notifications contain a `Type` attribute, whose value offers a way to categorize and filter notifications. The `Type` attribute in `MBeanServerNotification` contains only one of two possible strings: "JMX.mbean.registered" or "JMX.mbean.unregistered". JMX notifications also contain a `getType` method that returns the value of the `Type` attribute.

  The listener in Listing 7-3 invokes different lines of code depending on the value of the `Type` attribute.
- If a `JDBCDataSourceRuntimeMBean` has been registered, the listener passes the MBeans' object name to a custom method. The custom method registers a listener and configures a filter for the `JDBCDataSourceRuntimeMBean`; this MBean listener emits messages when the MBean's `Enabled` attribute changes.

  The implementation of the custom method is located in the **registration** class (not the filter class) so that the method can reuse registration class's connection to the MBean server. Such reuse is an efficient use of resources and eliminates the need to store credentials and URLs in multiple classes.

Listing 7-3 Example: Listening for MBeans Being Registered and Unregistered

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.MBeanServerNotification;
import javax.management.ObjectName;

public class DelegateListener implements NotificationListener {
   public void handleNotification(Notification notification, Object obj) {
      if (notification instanceof MBeanServerNotification) {
```

```
MBeanServerNotification msnotification =
(MBeanServerNotification) notification;

// Get the value of the MBeanServerNotification
// Type attribute, which contains either
// "JMX.mbean.registered" or "JMX.mbean.unregistered"
String nType = msnotification.getType();

// Get the object name of the MBean that was registered or
// unregistered
ObjectName mbn = msnotification.getMBeanName();

// Object names for WebLogic Server MBeans always contain
// a "Type" key property, which indicates the
// MBean's type (such as ServerRuntime or Log)
String key = mbn.getKeyProperty("Type");

if (nType.equals("JMX.mbean.registered")) {
    System.out.println("A " + key + " has been created.");

    System.out.println("Full MBean name: " + mbn);
    System.out.println("Time: " + msnotification.getTimeStamp());
       if (key.equals("JDBCDataSourceRuntime")) {
       // Registers a listener with a ServerRuntimeMBean.
       // By defining the "registerwithServerRuntime" method
       // in the "ListenToDelegate" class, you can reuse the
       // connection that "ListenToDelegate" established;
       // in addition to being an efficient way to use resources,
       // it eliminates the need to store credentials and URLs in
       // multiple classes.
       ListenToDelegate.registerwithJDBCDataSourceRuntime(mbn);
    }
}
if (nType.equals("JMX.mbean.unregistered")) {
    System.out.println("An MBean has been unregistered");
    System.out.println("Server name: " +
       mbn.getKeyProperty("Name"));
    System.out.println("Time: " + msnotification.getTimeStamp());
    System.out.println("Full MBean name: "
       + msnotification.getMBeanName());
}
```

```
        }
    }
}
```

Listing 7-4 shows methods from a registration class. Note the following:

- The JMX object name for `MBeanServerDelegate` is always `"JMImplementation:type=MBeanServerDelegate"`.
- The `main` method configures an instance of `javax.management.NotificationFilterSupport` to forward notifications only if value of the notification's Type attribute starts with "JMX.mbean.registered" or "JMX.mbean.unregistered".
- The `registerwithJDBCDataSourceRuntime` method registers the listener in Listing 7-1, "Notification Listener," on page 7-9 with the specified `JDBCDataSourceRuntimeMBean` instance. The method also configures a `javax.management.AttributeChangeNotificationFilter`, which forwards only `AttributeChangeNotification`s that describe changes to an attribute named `Enabled`.

To compile and run these methods, use the supporting custom methods from Listing 7-2 and run the resulting class as a remote JMX client.

Listing 7-4 Example: Registering a Listener with MBeanServerDelegate

```
public static void main(String[] args) throws Exception {
    String hostname = args[0];
    String portString = args[1];
    String username = args[2];
    String password = args[3];
    ObjectName delegate = new ObjectName(
        "JMImplementation:type=MBeanServerDelegate");

    try {
        //Instantiating your listener class.
        StartStopListener slistener = new StartStopListener();
        NotificationFilterSupport filter = new NotificationFilterSupport();
        filter.enableType("JMX.mbean.registered");
        filter.enableType("JMX.mbean.unregistered");
```

```
        /* Invoke a custom method that establishes a connection to the
         * Domain Runtime MBean Server and uses an instance of
         * MBeanServerConnection to represents the connection. The custom
         * method assigns the MBeanServerConnection to a class-wide, static
         * variable named "connection".
         */
        initConnection(hostname, portString, username, password);

        //Passing the name of the MBeans and your listener class to the
        //addNotificationListener method of MBeanServer.
        connection.addNotificationListener(delegate, slistener, 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);
     }
  }

  // Called by the listener if it receives notification of a
  // JDBCDataSourceRuntimeMBean being registered.
  public static void registerwithJDBCDataSourceRuntime(ObjectName mbname) {
     try {
        MyListener mylistener = new MyListener();
        AttributeChangeNotificationFilter filter =
           new AttributeChangeNotificationFilter();
        filter.enableAttribute("Enabled");

        connection.addNotificationListener(mbname, mylistener,
           filter, null);
     } catch (Exception e) {
        System.out.println("Exception: " + e);
     }
  }
```

Using Monitor MBeans to Observe Changes: Main Steps

To configure and use monitor MBeans:

1. Choose the type of monitor MBean type that supports your monitoring needs. "Monitor MBean Types and Notification Types" on page 7-21
2. Create a listener class that can listen for notifications from monitor MBeans. See "Creating a Notification Listener for a Monitor MBean" on page 7-23.
3. Create a class that creates, registers and configures a monitor MBean, registers your listener class with the monitor MBean, and then starts the monitor MBean. See "Registering the Monitor and Listener" on page 7-24

Monitor MBean Types and Notification Types

JMX provides monitor MBeans that are specialized to observe specific types of changes:

- `StringMonitorMBean` observes attributes whose value is a `String`.

  Use this monitor to periodically observe attributes such as `ServerLifeCycleRuntimeMBean State`.

  See `javax.management.monitor.StringMonitor` in the *J2SE 5.0 API Specification*, which implements `StringMonitorMBean`.

- `GaugeMonitorMBean` observes attributes whose value is a `Number`.

  Use this monitor to observe an attribute whose value fluctuates as a result of normal operations. Configure the gauge monitor to emit a notification if the value of the attribute fluctuates outside a specific range. For example, you can use it to monitor the `ThreadPoolRuntimeMBean StandbyThreadCount` attribute to verify that the number of unused but available threads in a server falls within an acceptable range.

  See `javax.management.monitor.GaugeMonitor` in the *J2SE 5.0 API Specification*, which implements `GaugeMonitorMBean`.

- `CounterMonitorMBean` observes attributes whose value is a `Number`.

  Use this monitor to observe an attribute whose value only increases as a result of normal operation. Configure the counter monitor to emit a notification if the value of the attribute crosses an upper threshold. You can also configure the counter monitor to increase the threshold and then reset the threshold at a specified point.

For example, to track the overall number of hits on a server and to be notified each time 100 additional hits have accumulated, use a counter monitor that observes the `ServerRuntimeMBean SocketsOpenedTotalCount` attribute.

See `javax.management.monitor.CounterMonitor` in the *J2SE 5.0 API Specification*, which implements `CounterMonitorMBean`.

All monitor MBeans emit notifications of type `javax.management.monitor.MonitorNotification`. When a monitor MBean generates a notification, it describes the event that generated the notification by writing a specific value into the notification's `Type` property. Table 7-4 describes the value of the `Type` property that the different types of monitor MBeans encode. A filter or listener can use the notification's `getType()` method to retrieve the `String` in the `Type` property.

Table 7-4 Monitor MBeans and the MonitorNotification Type Property

A Monitor MBean of This Type	Encodes This String in the MonitorNotification's Type Property
`CounterMonitor`	`jmx.monitor.counter.threshold` when the value of the counter reaches or exceeds a threshold known as the comparison level.
`GaugeMonitor`	• `jmx.monitor.gauge.high` if the observed attribute value is **increasing** and becomes equal to or greater than the high threshold value. 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. • `jmx.monitor.gauge.low` if the observed attribute value is **decreasing** and becomes equal to or less than the low threshold value. 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`	• `jmx.monitor.string.matches` if the observed attribute value **matches** the string to compare value. Subsequent matches of the string to compare values do not cause further notifications unless the attribute value differs from the string to compare value. • `jmx.monitor.string.differs` if the attribute value **differs** from the string to compare value. Subsequent differences from the string to compare value do not cause further notifications unless the attribute value matches the string to compare value.

Errors and the MonitorNotification Type Property

If an error occurs, all monitors encode one of the following values in the notification's `Type` property:

- `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 encode `jmx.monitor.error.threshold` into the `Type` property 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

When an observed attributes meets the criteria that you specify, a monitor MBean emits a notification. There are no special requirements for creating a listener for a `MonitorNotification`. The steps are the same as those described in "Creating a Notification Listener" on page 7-8 except:

- You listen for notifications of type `MonitorNotification`.
- Optionally, you can import the `javax.management.monitor.MonitorNotification` class and invoke its methods to retrieve additional information about the event that generated the notification.

See Listing 7-5.

Listing 7-5 Listener for Monitor Notifications

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

public class MonitorListener implements NotificationListener {
   public  void handleNotification(Notification notification, Object obj) {
      if(notification instanceof Notification) {
         Notification notif = (Notification) notification;
         System.out.println("Notification type" + notif.getType() );
         System.out.println("Message: " + notif.getMessage() );
      }
      if (notification instanceof MonitorNotification) {
         MonitorNotification mn = (MonitorNotification) notification;
         System.out.println("Observed Attribute: " +
             mn.getObservedAttribute());
         System.out.println("Trigger: " + mn.getTrigger() );
      }
   }
}
```

Registering the Monitor and Listener

Recall that to use a monitor MBean, you first must create and register an instance of the monitor MBean in the MBean server. Then you register a listener with the monitor MBean that you created. You can do all of this in a single class.

To register a monitor MBean, register your listener, and start the monitor MBean:

1. Initialize a connection to the Domain Runtime MBean Server.

   See "Make Remote Connections to an MBean Server" on page 4-2.

2. Create an `ObjectName` for your monitor MBean instance.

   See `javax.management.ObjectName` in the *J2SE 5.0 API Specification*.

   BEA recommends that your object name starts with the name of your organization and includes key properties that clearly identifies the purpose of the monitor MBean instance.

   For example, `mycompany:Name=SocketMonitor,Type=CounterMonitor`

3. Create and register one of the monitor MBeans.

Use `javax.management.MBeanServerConnection.createMBean(String` *`classname`* `ObjectName` *`name`*`)` method where:

- *`classname`* is one of the following values:
  `javax.management.monitor.CounterMonitor`
  `javax.management.monitor.GaugeMonitor`
  `javax.management.monitor.StringMonitor`
- *`name`* is the object name that you created for the monitor MBean instance.

4. Configure the monitor MBean by setting the value of its attributes.

   For guidelines on which attributes to set, see the `javax.management.monitoring` package in the *J2SE 5.0 API Specification*.

5. To specify the MBean that your monitor MBean monitors (the observed MBean), invoke the monitor MBean's `addObservedObject(ObjectName` *`objectname`*`)` and `addObservedAttribute(String` *`attributename`*`)` operations where.

   - *`objectname`* is the `ObjectName` of the observed MBean
   - *`attributename`* is the name of the attribute in the observed MBean that you want to monitor

   A single instance of a monitor MBean can monitor multiple MBeans. Invoke the `addObservedObject` and `addObservedAttribute` operation for each MBean instance that you want to monitor.

6. Instantiate the listener object that you created in "Creating a Notification Listener for a Monitor MBean" on page 7-23.

7. Optionally instantiate and configure a filter.

8. Register the listener and optional filter with the **monitor MBean**. Do not register the listener with the observed MBean.

   Invoke the monitor MBean's `addNotificationListener(NotificationListener` *`listener`*`, NotificationFilter` *`filter`*`, Object` *`handback`*`)` method.

9. Start the monitor by invoking the monitor MBean's `start()` operation.

Example: Registering a CounterMonitorMBean and Its Listener

Listing 7-6 shows the `main()` method of a class that creates and configures a `CounterMonitorMBean` to observe the `SocketsOpenedTotalCount` attribute in each

ServerRuntimeMBean instance in a domain. (See SocketsOpenedTotalCount in *WebLogic Server MBean Reference*.)

The code example connects to the Domain Runtime MBean Server so that it can monitor multiple instances of ServerRuntimeMBean. Note the following:

- Only one instance of CounterMonitorMBean monitors all instances of ServerRuntimeMBean. The Domain Runtime MBean Server gives the CounterMonitorMBean federated access to instances of ServerRuntimeMBean that are running in a different JVM.
- Only one instance of your listener class and the filter class listens and filters notifications from the CounterMonitorMBean.

To compile and run this main method, use the supporting custom methods from Listing 7-2 and run the resulting class as a remote JMX client.

Listing 7-6 Example: Registering a CounterMonitorMBean and Its Listener

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

   try {
      /* Invokes a custom method that establishes a connection to the
       * Domain Runtime MBean Server and uses an instance of
       * MBeanServerConnection to represents the connection. The custom
       * method assigns the MBeanServerConnection to a class-wide, static
       * variable named "connection".
       */
      initConnection(hostname, portString, username, password);

      //Creates and registers the monitor MBean.
      ObjectName monitorON =
         new ObjectName("mycompany:Name=mySocketMonitor,Type=CounterMonitor");
      String classname = "javax.management.monitor.CounterMonitor";
      System.out.println("===> create mbean "+monitorON);
      connection.createMBean(classname, monitorON);

      //Configure the monitor MBean.
      Number initThreshold = new Long(2);
      Number offset = new Long(1);
      connection.setAttribute(monitorON,
         new Attribute("InitThreshold", initThreshold));
```

```
      connection.setAttribute(monitorON, new Attribute("Offset", offset));
      connection.setAttribute(monitorON,
         new Attribute("Notify", new Boolean(true)));

      //Gets the object names of the MBeans that you want to monitor.
      ObjectName[] serverRT = getServerRuntimes();
      int length= (int) serverRT.length;
      for (int i=0; i < length; i++) {
         //Sets each instance of ServerRuntime MBean as a monitored MBean.
         System.out.println("===> add observed mbean "+serverRT[i]);
         connection.invoke(monitorON, "addObservedObject",
            new Object[] { serverRT[i] },
            new String[] { "javax.management.ObjectName" });

         Attribute attr = new Attribute("ObservedAttribute",
            "SocketsOpenedTotalCount");
         connection.setAttribute(monitorON, attr);
      }

      // Instantiates your listener class and configures a filter to
      // forward only counter monitor messages.
      MonitorListener listener = new MonitorListener();
      NotificationFilterSupport filter = new NotificationFilterSupport();
      filter.enableType("jmx.monitor.counter");
      filter.enableType("jmx.monitor.error");

      //Uses the MBean server's addNotificationListener method to
      //register the listener and filter with the monitor MBean.
      System.out.println("===> ADD NOTIFICATION LISTENER TO "+monitorON);
      connection.addNotificationListener(monitorON, listener, filter, null);
      System.out.println("\n[myListener]: Listener registered ...");

      //Starts the monitor.
      connection.invoke(monitorON, "start", new Object[] { }, new String[] { });

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

CHAPTER 8

Configuring WebLogic Server JMX Services

Within a WebLogic Server domain, you can specify which JMX services are available. For example, in a production environment you can disable the WebLogic Server editing service and therefore prevent most runtime changes to the domain.

The following attributes of `JMXMBean` determine which JMX services are available in a domain (see `JMXMBean` in *WebLogic Server MBean Reference*):

- `EditMBeanServerEnabled` controls whether JMX clients, including utilities such as the Administration Console and the WebLogic Scripting Tool, can modify a domain's configuration.
- `DomainMBeanServerEnabled` controls whether JMX clients can access all runtime MBeans and read-only configuration MBeans through a single connection to the Domain Runtime MBean Server.
- `RuntimeMBeanServerEnabled` controls whether JMX clients can access a specific server's runtime MBeans and read-only configuration MBeans through the server's Runtime MBean Server.
- `PlatformMBeanServerEnabled` controls whether all WebLogic Server instances start their Runtime MBean Servers as the JDK platform MBean server. This makes it possible to access WebLogic Server MBeans and the JVM platform MBeans from a single MBean server.
- `CompatibilityMBeanServerEnabled` enables JMX clients to use the deprecated `weblogic.management.MBeanHome` interface to access WebLogic Server MBeans.

- `ManagementEJBEnabled` controls whether the current WebLogic Server domain supports the J2EE Management APIs.

Example: Using WebLogic Scripting Tool to Make a Domain Read-Only

The following example uses the WebLogic Scripting Tool (WLST) to set the `JMXMBean EditMBeanServerEnabled` attribute to `false`. It assumes that you are running WLST on a Windows computer, that you created a domain under `c:\mydomain`, and that you have not deleted the scripts that WebLogic Server creates along with your domain.

Caution: The following steps prevent JMX clients (including the WebLogic Server Administration Console and the WebLogic Scripting Tool in online mode) from modifying the domain's configuration. You can still modify the domain configuration through the offline editing feature of WebLogic Scripting Tool.

These steps do not prevent JMX clients from deploying or undeploying modules because the WebLogic Server deployment service does not use JMX.

1. Start the domain's Administration Server.
2. In a command prompt, set up the required environment by running the following script:

   ```
   c:\mydomain\setDomainEnv.cmd
   ```

3. In the same command prompt, enter the following commands:

   a. `java weblogic.WLST`

   b. `connect('weblogic','weblogic')`

   c. `edit()`

   d. `startEdit()`

   e. `cd('JMX/mydomain')`

   f. `set('EditMBeanServerEnabled','false')`

   g. `activate()`

   h. `exit()`

Index

P

R

S

T