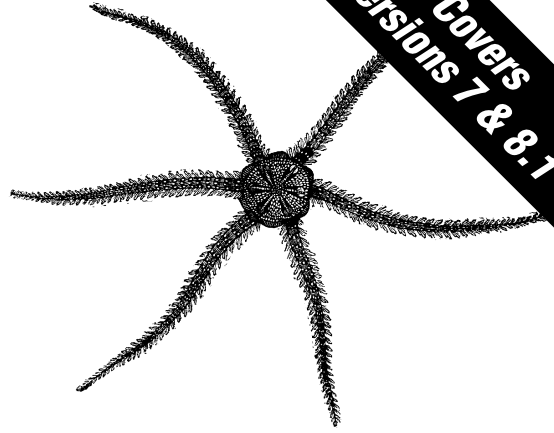


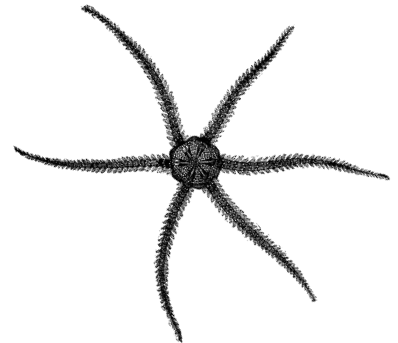
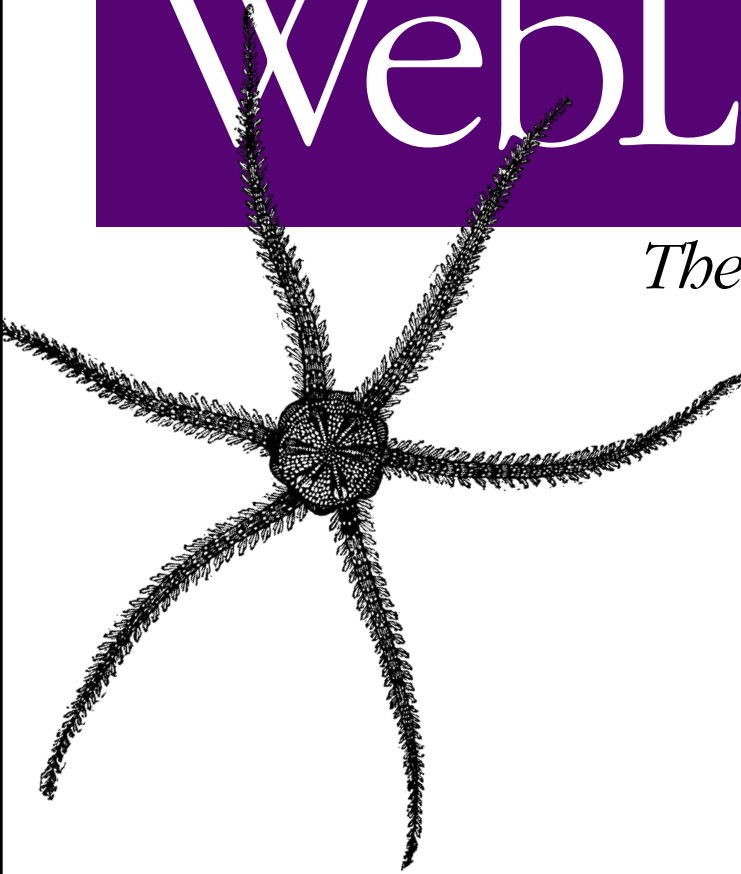
*Development, Deployment & Maintenance*

**Covers  
Versions 7 & 8.1**



# WebLogic™

*The Definitive Guide*



**O'REILLY®**

*Jon Mountjoy & Avinash Chugh*

The Java Management Extensions (JMX) specification defines the architecture, services, and API for the distributed management of resources using Java. JMX can be used to instrument everything from network hardware to applications, enabling you to build your own applications that manage these instrumented resources. This chapter focuses on how WebLogic Server is itself instrumented, and how this enables you to create applications that monitor and manage various aspects of a WebLogic domain and its deployed applications.

WebLogic's JMX implementation, and the specification itself, comprise three levels: an instrumentation, agent, and distribution level. The *instrumentation level* provides a design for implementing JMX-manageable resources. Within WebLogic, the manageable resources include just about everything, ranging from connection pools and security realms to the domain configuration and the state of a deployed application. The instrumentation of a resource is provided by MBeans, which expose an interface for the management and control of that resource. For instance, the MBeans for a JDBC connection pool expose attributes such as the pool's name and size, as well as operations such as resetting the pool or shutting it down. Runtime statistics are also made available, such as the maximum number of connections to the pool and the connection delay time.

The *agent level* builds upon the instrumentation level to provide a standardized way of managing MBeans. In WebLogic, this is realized by an MBean Server, which hosts the MBeans on an individual WebLogic instance and lets clients access, retrieve, and modify MBeans on the MBean Server. You typically would write management applications that interact with the MBean Server and its hosted MBeans. For example, you can write an application that locates all JDBC connection pool MBeans and ensures that they don't have unacceptable connection time delays. WebLogic supports two ways of interacting with MBeans. You could either adopt the standard approach put forward in the JMX specification or use a WebLogic-specific type-safe interface. The standard approach provides a generic interface for accessing any MBean. For example, in order to invoke an operation on an MBean, you need to pass

the operation name and an array of arguments to an invocation method. WebLogic's type-safe implementation exposes an individual interface for each MBean, thereby making manipulating MBeans as easy as using traditional JavaBeans.

As WebLogic domains are distributed, the manageable resources are themselves spread across a number of server instances. WebLogic's JMX architecture provides each WebLogic instance with its own MBean Server. WebLogic's *distribution level* defines precisely how MBeans are made accessible through this distributed management architecture. For example, when a Managed Server instance starts up, it contacts the Administration Server for a set of MBeans that describes its local configuration. The Managed Server then uses these local configuration MBeans to establish the services that have been configured for the server.

The JMX specification provides two powerful additions to this framework that allow you to build effective management applications. The MBean notification model allows MBeans to broadcast management events, called *notifications*. By registering a listener class with an MBean, you can be alerted when a notification has been broadcast and can react to it appropriately. The Monitor MBean architecture provides the second management enhancement. This architecture defines a set of MBeans that monitor other MBeans, observing specific attribute values as they vary over time and automatically firing notifications when these values exceed specified thresholds. For instance, you could configure a monitor MBean to observe the connection count value on a connection pool and alert you when the pool exceeds specified thresholds.

All of these facilities provide you with the means to create applications that enable you to remotely configure, monitor, and manage the different aspects of your WebLogic domain and the applications deployed to it. You can programmatically monitor and manipulate a WebLogic deployment, leading to exciting opportunities. For instance, a management application could constantly monitor the status of JMS servers, and in the case of failure, automatically migrate the service to an alternate WebLogic Server—a service not provided by the standard JMS tooling in WebLogic. The Administration Console is itself an example of a remote management application. In fact, it is nothing more than a web-based tool that allows Administrators and Deployers to remotely manipulate MBeans.

WebLogic has literally hundreds of instrumented services, and it would be senseless to explain each one in detail. In addition, any deployments such as EJBs or servlets become manageable resources as well. Instead of looking at each MBean in detail, something that is covered by the JavaDoc APIs, this chapter will teach you how to effectively use WebLogic's JMX implementation. You will learn about WebLogic's distributed JMX architecture, and how to access and manipulate MBeans to configure, manage, and monitor a wide range of managed services. En route, a number of code examples will demonstrate how to use particular MBeans. You will also see an example of using the `WLShell` tool, which provides a scripting environment for navigating the MBeans in a running WebLogic instance. You can use this tool to script MBean actions, such as modifying a domain configuration, or simply to view MBean attributes and invoke operations.

# The MBean Architecture

Every WebLogic instance owns an MBean Server that hosts a number of MBeans. The MBean Server acts as a registry for the MBeans and provides services for accessing and manipulating MBeans running on the server. Because a WebLogic domain may be distributed across multiple machines with differing deployments and domain resources targeted to different servers, the MBean Server for each WebLogic instance will hold different types of MBeans. For instance, the runtime statistics for a JDBC connection pool on a server can be obtained only from an MBean running on that server. WebLogic makes a distinction between three different sets of MBeans:

## *Configuration MBeans*

These expose attributes and operations for the configuration of a resource.

## *Runtime MBeans*

These provide information about the runtime state of a resource.

## *Security MBeans*

These reflect the SSPIs, providing direct access to the configuration of WebLogic's security framework.

To fully understand the MBean architecture as implemented in WebLogic, you need to understand the different types of MBeans and how WebLogic distributes these MBeans across MBean Servers within a domain.

## Configuration MBeans

A Configuration MBean holds the configuration for a managed resource or service within a WebLogic domain. WebLogic captures the configuration settings for all kinds of domain resources and services: web servers, clusters, JDBC connection pools, J2EE components (EJBs, web applications, resource adapters), JMS destinations, XML Registries, web services, and many more. The definitive list of Configuration MBeans can be found within the API documentation under the `weblogic.management.configuration` package. Here are a few examples:

### `DomainMBean`

This MBean represents a WebLogic Domain. You can use it to access various attributes of the domain (e.g., Administration Port), or even explore subcategories of configuration settings such as JTA, Security, Logging, and others. In addition, you can access the configuration settings for the managed servers within the domain.

### `ServerMBean`

This MBean represents a WebLogic Server instance. You can use it to access virtually all the attributes for the server: server name, listen address and port, external DNS name, and many more. You also can use the MBean to acquire configuration settings for the web server, the cluster to which the server belongs, the machine that hosts the server, the XML Registry, and other related resources that have been configured for the server.

## JDBCConnectionPoolMBean

This MBean represents a JDBC connection pool configured within a domain. It can be used to access the various configuration settings for a connection pool, such as its capacity, driver name, and database URL.

The configuration of a managed resource is described entirely by the state of the MBean representing that resource. As we learned in Chapter 13, the Administration Server is responsible for managing the configuration of a domain and distributing the relevant portions of the configuration to different Managed Servers. The *config.xml* file that is used by the Administration Console to persist the current state of the domain's configuration is nothing more than a serialization of all the Configuration MBeans on the Administration Server. When the Administration Server starts up, it reads this domain configuration file and reconstructs the set of Configuration MBeans in its local MBean Server. So, the Administration Server is the prime host for all Configuration MBeans, and using the Administration Console is one of the easiest ways of manipulating these MBeans.

When a Managed Server starts up, it asks the Administration Server for its own server configuration. The Administration Server responds by sending back a subset of those MBeans that are relevant to the Managed Server. This includes the MBeans that reflect the configuration of all resources and services deployed to that server. Hence, the Administration Server holds the master copy of all Configuration MBeans for all resources within a domain, while the MBean Server on each Managed Server hosts local replicas for all resources that are targeted to the Managed Server.

This replication of Configuration MBeans hosted on the Administration Server occurs for performance reasons as well. Local clients of the Managed Server can simply use the local replicas of the Configuration MBeans, without having to contact the MBean Server running on the Administration Server. Figure 20-1 illustrates this process.

The Configuration MBeans hosted by the Administration Server are called *Administration MBeans*, while the replicas hosted on the Managed Servers are called *Local Configuration MBeans*. Any permanent changes to the configuration of a domain must be made through the Administration MBeans. These changes then will propagate through to the Local Configuration MBeans on all running Managed Servers.

The Administration Server periodically serializes the state of its Administration MBeans to the domain's *config.xml* configuration file, making them permanent. This serialization also occurs when the Administration Server detects that one or more configuration settings have been changed. Sometimes, a domain resource instantly is updated to reflect any changes made to its configuration settings. In other cases, the resource may define configuration settings whose value changes come into effect only after you restart the server that hosts the resource. In this case, the changes to the configuration won't dynamically alter the behavior of a resource. The Administration Console gives you an indication of which configuration settings require a

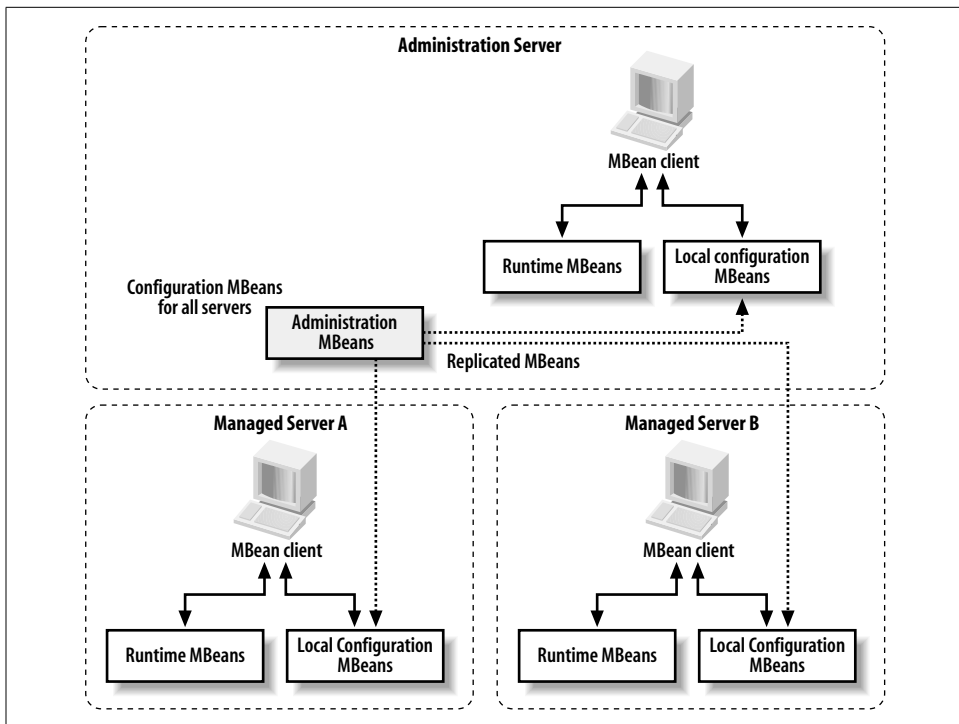


Figure 20-1. Distribution of Runtime and Configuration MBeans in a domain

server restart by placing a caution icon (a yellow triangular icon) next to the attribute name. Although the changes to these attributes are persisted instantly, the actual change to the resource occurs only after you restart the server to which the resource or service has been targeted. The JavaDoc documentation also indicates explicitly which attributes of the MBean are dynamically configurable.

Though Administration MBeans have a persisted state, the Local Configuration MBeans are transient. If the server instance hosting the Local MBeans is shut down, then their in-memory state is lost. This is not a problem because the Local MBeans are reinitialized the next time the server boots and makes contact with the Administration Server. You can, however, make temporary changes to particular server instances. For example, you could use the command-line option `-Dweblogic.ListenPort=7501` during server startup to change the listen port of a Managed Server. This option temporarily updates the listen port information in the local `ServerMBean` running on that Managed Server only, and the change lasts as long as the server is alive. Moreover, this setting has no effect on the master copy of the `ServerMBean` on the Administration Server.

The same distinction between Local and Administration MBeans is maintained even if you deploy applications to the Administration Server. In this case, the Administration Server hosts both Administration and Local Configuration MBeans.

Chapter 13 shows how a Managed Server can be configured to start up in Managed Service Independence (MSI) mode, in the absence of an Administration Server. This mode effectively replicates the domain's configuration file; when the Managed Server starts up, it creates its own copy of the Administration MBeans. It then extracts its own Local Configuration MBeans, allowing it to start up as usual. You should not interact with the Administration MBeans on a server running in MSI mode, as it does not host the definitive configuration, and any changes you make will not be replicated to the rest of the domain.

## Runtime MBeans

Runtime MBeans reflect the runtime state of resources. As such, they are not replicated like Configuration MBeans, but rather exist only on the same server as the underlying managed resource. Figure 20-1 shows how each server instance holds its own set of Runtime MBeans and Local Configuration MBeans. WebLogic maintains the runtime state of a whole range of domain resources, including servers, clusters, EJB instance pools, JTA transactions, servlets, JMS producers and subscribers, and many more. The definitive list of Runtime MBeans can be found in the API documentation under the `weblogic.management.runtime` package. Here are a few examples of Runtime MBeans provided by WebLogic:

### ServerRuntimeMBean

This MBean holds the runtime state of a WebLogic instance. It can be used to find the state of the server instance and the number of open sockets.

### EJBPoolRuntimeMBean

This MBean can be used to access the runtime state of an EJB pool, providing access to an array of runtime statistics, including the total number of bean instances in use and the number of threads waiting for a bean instance to be made available.

### JDBCConnectionPoolRuntimeMBean

This MBean can be used to access the runtime information of a JDBC connection pool, providing access to the total number of current connections.

A Runtime MBean provides runtime information on a resource, as well as operations that can modify its runtime state. For instance, the `ServerRuntimeMBean` allows you to restart a suspended server. Runtime MBeans are transient, so when the hosting server instance is shut down, all of the MBean's associated state is lost.

## Security MBeans

The third category of MBeans reflects the SSPIs provided by WebLogic's security framework, which we covered earlier in Chapter 17. These MBeans can be found in the `weblogic.management.security` package hierarchies. The security framework uses the MBeans to access and configure the security within a domain, so any implementation of the SSPIs must be accompanied by an implementation of the MBeans as

well. Naturally, the default security implementation that ships with your WebLogic distribution does just this.

The Security MBeans comprise sets of required and optional MBeans. The required MBeans provide access to the main interfaces. For instance, the `AuthenticatorMBean` is used by the authentication services. The base MBean implementation merely lets you retrieve and set the control flag setting. Custom implementations of the Authenticator SSPI also are required to provide an MBean with which to manage the implementation. This custom Security MBean must extend the `AuthenticatorMBean`, and may even provide additional management attributes and operations. Security SSPI implementations may implement additional MBean interfaces, such as the `UserAddEditorMBean` and `UserRemoverMBean` that can be used to create, edit, and remove users from the security realm.

## Accessing MBean Servers

Now that you know how WebLogic organizes MBeans across MBean Servers, we can turn to accessing MBean Servers. The first point of entry is to find a home interface for the MBean Server. The home interface provides access to the underlying MBean server and enables you to reach the MBeans hosted on the server.

### Local and Administration Home Interfaces

The MBean Server running on a WebLogic Server instance can be reached through the `weblogic.management.MBeanHome` interface. Two implementations of this home interface can be obtained:

#### *Local Home interface*

This interface provides access to the local MBeans hosted on the MBean Server itself. This means that you can use the Local Home interface to access the Local Configuration and Runtime MBeans on that server instance, but not interact with any of the Administration MBeans.

#### *Administration Home interface*

This interface, exposed only by the Administration Server, provides access to the Administration MBeans and all other MBeans on all other server instances within the domain.

Figure 20-2 illustrates this architecture.

As indicated in Figure 20-2, you also can reach the Runtime MBeans on any Managed Server through the Administration Home interface itself, instead of using the Local Home interface of the Managed Server directly. Clearly, it is slower to access the local MBeans through the Administration interface rather than to access them directly through the Local interface. You incur an additional performance penalty due to the overhead imposed by the RMI communication with the local server

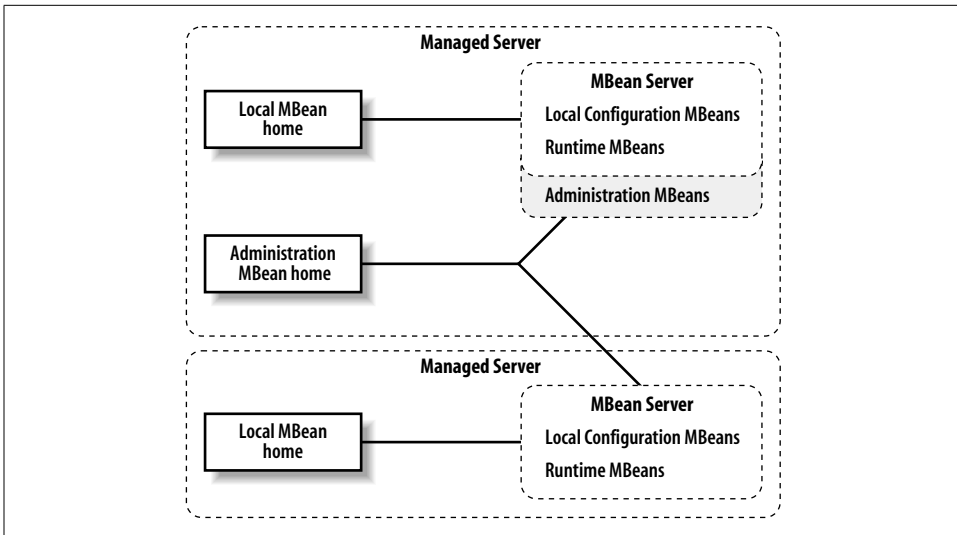


Figure 20-2. Home interfaces and their hosted MBeans

MBeans. If you need to manipulate the local MBeans of a particular Managed Server, you should contact that server directly through its Local Home interface.

## Retrieving the Home Interface

The home interfaces can be retrieved either by looking up the server's JNDI tree or by using a WebLogic-specific helper class. In both cases, you need to supply the authentication details of a user who is authorized to access the domain resource and the URL of the server that you wish to contact. If you intend to access a local home, you also need the name of the server that you would like to contact.

Use the `getMBeanHome()` method on the `weblogic.management.Helper` class to access the Local MBean home, and use the `getAdminMBeanHome()` method on the `Helper` class to access the Administration MBean home. The following code sample shows how to use these helper methods to retrieve the Local and Administration home interfaces:

```
public MBeanHome findLocalHomeHelper(String url, String serverName) {
    weblogic.management.MBeanHome localHome = null;
    localHome =
        (MBeanHome) Helper.getMBeanHome(USERNAME, PASSWORD, url, serverName);
    return localHome;
}

public MBeanHome findAdminHomeHelper(String adminURL) {
    weblogic.management.MBeanHome adminHome = null;
    adminHome =
        (MBeanHome) Helper.getAdminMBeanHome(USERNAME, PASSWORD, adminURL);
    return adminHome;
}
```

Now, if you need to access the Local home of ServerB, you need to simply issue the following call:

```
MBeanHome localHomeB = findLocalHomeHelper("t3://10.0.10.14:7001", "ServerB");
```

Using the JNDI to obtain a home is just as easy. The locations of the Local and Administration homes are stored in the constants `MBeanHome.LOCAL_JNDI_NAME` and `MBeanHome.ADMIN_JNDI_NAME`, respectively. Once you establish an initial context with a server, you simply look up the home interface using these names:

```
InitialContext ctx = new InitialContext();
MBeanHome localHomeB = (MBeanHome) ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);
```

An initial JNDI context obtained from the Administration Server also allows you to access the Local home from any of the Managed Servers. In this case, the JNDI name used to look up the Local home matches the following string:

```
"weblogic.management.home" + "." + ServerName
```

This prefix `weblogic.management.home` is conveniently bound to the constant `MBeanHome.JNDI_NAME`. The following code sample shows how to locate the Local Home interface of ServerB via the initial JNDI context obtained from the Administration Server:

```
//create an initial JNDI context using the URL of the Admin Server
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, "t3://adminserver:8001");
//set other environment properties for the initial JNDI Context
InitialContext ctx = new InitialContext(env);
//now look up Local home by prefixing MBeanHome.JNDI_NAME to "ServerB"
MBeanHome localHomeB = (MBeanHome) ctx.lookup(MBeanHome.JNDI_NAME + "." + "ServerB");
```

## Accessing MBeans

Once you obtain a home interface, there are several ways to access the underlying MBean Server. The first approach is to use the `MBeanHome` interface to retrieve an instance of `javax.management.MBeanServer`, the standard JMX interface for interacting with an MBean Server. It provides a generic way for accessing the attributes and invoking the operations exposed by an MBean. The onus is on you to supply the correct number of parameters, of the correct type, when you use an `MBeanServer` instance to issue method calls to an MBean.

The second approach is to use a proprietary type-safe interface, which is implemented as a wrapper around the `MBeanServer` interface. At the cost of portability, you gain very simplified and compact access to MBeans and their attributes and operations. WebLogic's Administration Tool provides another nonprogrammatic approach to accessing MBeans. In addition, the new `WLSHELL` tool provides a shell environment for accessing and manipulating MBeans on a particular WebLogic instance.

## Naming of MBeans

Before you can use the `MBeanServer` interface to locate MBeans, you need to understand how WebLogic names its MBeans. Each MBean hosted by an MBean Server is uniquely named, and every MBean name is constructed using an instance of the `JMX ObjectName` class. WebLogic uses instances of the `WebLogicObjectName` class, which extends `ObjectName` to carry additional information. When printed, the name of an MBean follows this format:

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

So, a name contains a domain name, followed by an unordered list of property/value pairs, explained here:

- A domain name identifies the domain to which the MBean belongs. For most MBeans, this is just the name of the WebLogic administration domain. For Security MBeans, the name of the domain must be Security.
- A Name property identifies the name of the resource associated with the MBean. For instance, if you've created a JMS server with the name My JMS server, the name used to locate the associated MBean may use the `Name=My JMS Server` property. Note that this property does not represent the JNDI name to which the resource is bound.
- A Type property points to the interface implemented by the MBean's class. It also indicates whether the MBean is an Administration, Local Configuration, or Runtime MBean.
- A Location property identifies the name of the server instance on which the MBean is running. You don't need to specify this property when locating Administration MBeans.
- MBeans that are in a parent-child relationship with other MBeans use an additional `TypeOfParentMBean=NameOfParentMBean` property/value pair to express this relationship.

For example, if you defined a data source in the `myClusterDomain` domain, the name of the associated Administration MBean is:

```
myClusterDomain:Name=My Data Source,Type=JDBCDataSource
```

To construct a name object representing such an MBean, use the three-argument `WebLogicObjectName` constructor, which takes the name, the type, and the domain of the object:

```
WebLogicObjectName oname = new WebLogicObjectName("My Data Source",  
                                                    "JDBCDataSource", "myClusterDomain");
```

## Determining MBean type

Every MBean within WebLogic is an instance of a class that implements one of the `weblogic.management.configuration` or `weblogic.management.runtime` interfaces.

When constructing the MBean's name, the value for the Type property that you need to supply is determined by a mangling of the name of the interface implementing either of the aforementioned interfaces.

For WebLogic's Runtime MBeans, the value for the Type property corresponds to the name of the MBean interface, but without the MBean suffix. So, for the `ServletRuntimeMBean`, the MBean's name must include the name/value pair `Type=ServletRuntime`. Similarly, for the `JDBCDataSourceMBean`, you must specify `Type=JDBCDataSource`.

For WebLogic's Configuration MBeans, the value for the Type property that you supply indicates whether the name refers to an Administration or a Local Configuration MBean. For an Administration MBean, the type corresponds to the name of the MBean's interface but, as before, without the MBean suffix. For a Local Configuration MBean, its type is obtained by appending `Config` to the type of its Administration MBean counterpart. For example, the MBean name for a `ServerMBean` on the Administration Server must use the name/value pair `Type=Server`. If you need to refer to a Local Configuration MBean, you must specify `Type=ServerConfig`.

Let's assume that our `myClusterDomain` domain has an Administration Server and two Managed Servers, `ServerA` and `ServerB`. If a data source is targeted to both Managed Servers, the names of the Local Configuration MBeans associated with the data source will be:

```
myClusterDomain:Location=ServerA,Name=My Data Source,Type=JDBCDataSourceConfig
myClusterDomain:Location=ServerB,Name=My Data Source,Type=JDBCDataSourceConfig
```

To construct a name object representing such an MBean, which has a location, use the four-argument `WebLogicObjectName` constructor, which takes the object's name, type, domain, and location:

```
WebLogicObjectName oname = new WebLogicObjectName("My Data Source",
    "JDBCDataSourceConfig", "myClusterDomain", "ServerA");
```

## Determining parent-child relationships

Except for the `DomainMBean`, all MBeans inherit either directly or indirectly from the `DomainMBean`. Sometimes you need to explicitly state this inheritance when constructing the MBean's name in order to uniquely identify it. For instance, consider the `LogMBean`, which represents the configuration of a log file. Log configuration can occur at several levels. At the domain level, you could set up the domain log configuration. At a server instance level, you could adjust the server log's configuration. Different instances of the `LogMBean` represent the configuration at different levels. You can differentiate between these configuration MBeans by indicating the fact that the server log Configuration MBean is a child of the server Configuration MBean running within the domain. The domain log Configuration MBean is an implicit child of `DomainMBean` and *not* a child of the server Configuration MBean.

To emphasize this point, let's take a look at an excerpt from the domain's *config.xml* configuration file:

```

<Domain ConfigurationVersion="8.1.0.0" Name="myDomain">
  <!-- etc. etc. -->
  <Log FileName=".\\wl-domain.log" Name="myDomain"/>
  <Server Name="Admin" ListenAddress="10.0.10.10" ListenPort="8001">
    <!-- etc. etc. etc. -->
    <Log FileName="Admin\\Admin.log" Name="Admin"/>
  </Server>
  <Server Name="ServerA" Cluster="MyCluster" InterfaceAddress="10.0.10.10"
    ListenAddress="10.0.10.10" ListenPort="7001">
    <!-- etc. etc. etc. -->
    <Log Name="ServerA"/>
  </Server>
  <Server Name="ServerB" Cluster="MyCluster" InterfaceAddress="10.0.10.14"
    ListenAddress="10.0.10.14" ListenPort="7001">
    <!-- etc. etc. etc. -->
    <Log Name="ServerB"/>
  </Server>
</Domain>

```

Here we can clearly see the log configuration for the domain and the two Managed Servers within the domain. The domain LogMBean does not need to express a relationship with the domain. It is unambiguous. The server's LogMBean instance does have to express a parent-child relationship. In this case, the parent is the ServerMBean instance associated with its Managed Server. The relationship between the Log MBeans for the domain and for the Administration Server is expressed in Figure 20-3.

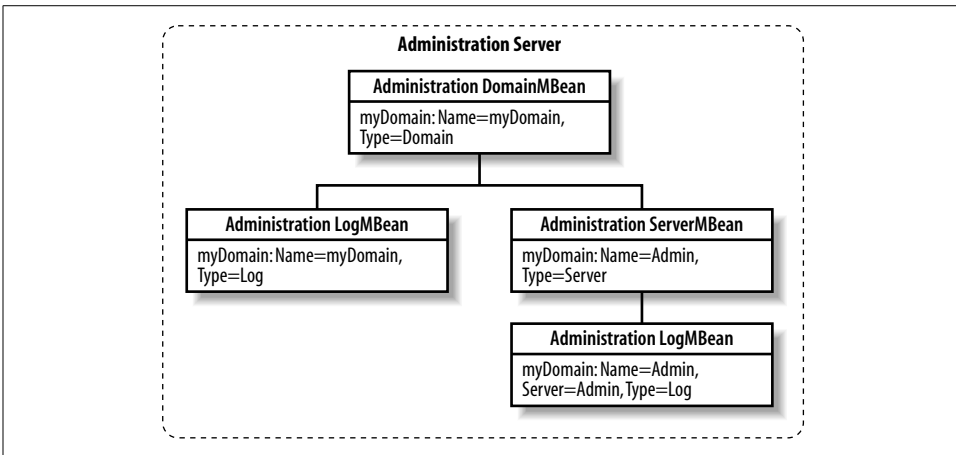


Figure 20-3. Relationships in the MBean hierarchy

There are two additional points to note from this figure.

- The child MBeans use the same name as the parent MBean, unless there are multiple children of the same type. Thus, the name of the domain's LogMBean is myDomain because the name of the parent MBean also is myDomain.
- The server's LogMBean is in a parent-child relationship, so its name also must include a *TypeOfParentMBean=NameOfParentMBean* property/value. In this case, we have an additional Server=Admin part to the LogMBean's name so that we can identify the parent MBean.

To construct a name object representing such an MBean, first create one for the parent and then another for the child. As shown in Figure 20-3, the parent MBean is of type ServerMBean. Constructing the name object for the parent uses the constructor that we have seen before:

```
WebLogicObjectName parent =
    new WebLogicObjectName("Admin", "Server", "myDomain");
```

Now, to construct the name of the child, we use the constructor that additionally takes the name of a parent. This time, of course, the type of the MBean is Log:

```
WebLogicObjectName childlog =
    new WebLogicObjectName("Admin", "Log", "myDomain", parent);
```

Though the MBean naming schema may seem tedious, it often makes it a lot easier to find the MBean you want either programmatically or by using the Administration Tool.

## Using the Administration Tool

The `weblogic.Admin` tool can be used to view and manipulate JMX data. You can use this tool for quick, nonprogrammatic access to the MBeans. WebLogic's Administration Tool supports three options pertinent to manipulating MBeans: the GET, SET, and INVOKE options. The GET option can be used to retrieve attribute values of an MBean, the SET to modify attribute values, and the INVOKE to invoke any operation on the MBean. Each command provides two ways in which you can identify the MBean on which to operate. The `-mbean` argument requires you to specify the name of the MBean, and the `-type` option requires you to specify the MBean's type.

The following example shows how to use the GET option to retrieve the attribute values of a named DataSource MBean:

```
java weblogic.Admin -url http://10.0.10.10:7001 -username system -password psst
  GET -pretty -mbean
  "myClusterDomain:Location=ServerA,Name=My DataSource,
  Type=JDBCDataSourceConfig"
```

This outputs the full list of attributes and values:

```
MBeanName: "myClusterDomain:Location=ServerA,Name=My DataSource,
  Type=JDBCDataSourceConfig"
  CachingDisabled: true
  ConnectionWaitPeriod: 1
```

```
DeploymentOrder: 1000
JNDIName: DS
Name: MyJDBC Data Source
Notes:
ObjectName: MyJDBC Data Source
PoolName: MyJDBCTool
Registered: false
RowPrefetchEnabled: false
RowPrefetchSize: 48
StreamChunkSize: 256
Targets: ServerB,MyCluster
Type: JDBCDataSourceConfig
WaitForConnectionEnabled: false
```

The tool can also help you determine the name of an MBean if you know its type only. The following example prints out the configuration settings encapsulated by the Log MBeans, which we explored in the previous section:

```
java weblogic.Admin -url http://10.0.10.10:7001 -username system -password pssst
GET -pretty -type Log
```

In our example domain, which consists of an Administration Server and two Managed Servers, this command outputs the settings for four Configuration Log MBeans, one for the domain Log MBean and one for each Log MBean associated with the servers.

## Using WLS Shell

BEA's dev2dev web site contains several tools for manipulating and viewing MBean data. The most flexible is *WLS Shell*, a powerful shell environment that lets you script actions to create, view, monitor, or modify MBeans. You will need to download and install the tool before you can use it. The latest version, together with many useful scripts, can be obtained from <http://www.wlshell.com/>.

After starting the tool, you will be presented with a command prompt that allows you to enter commands using the shell's scripting language. *WLS Shell*'s scripting language uses a filesystem analogy, whereby directories correspond to MBeans. In this analogy, creating a new directory is akin to creating a new MBean. It also provides the familiar `get`, `set`, and `invoke` commands on MBeans, as well as flow control and various other features. The following script illustrates these concepts. It creates a new connection pool, configures the relevant attributes of the pool, and assigns it to a server:

```
connect 10.0.10.10:7001 system password

targetServer = /Server/MyServer

mkdir /JDBCConnectionPool/MyJDBCTool
cd /JDBCConnectionPool/MyJDBCTool
set DriverName "com.microsoft.jdbcx.sqlserver.SQLServerDataSource"
set InitialCapacity 5
```

```

set MaxCapacity 10
set PreparedStatementCacheSize 15
set Properties (Properties) "user=sa;url=jdbc:microsoft:sqlserver://
;password=pw;PortNumber=143;SelectMethod=cur
sor;ServerName=10.0.10.10;dataSourceName=IsiPool;DatabaseName=Galactica"
set SupportsLocalTransaction true
set TestConnectionsOnRelease false
set URL "jdbc:microsoft:sqlserver://"
invoke addTarget $targetServer
cd /

```

The following script retrieves a few MBean attribute values and then invokes the stop operation on the Server MBean:

```

get -r ServerRuntime/myserver/State
get JTARuntime/JTARuntime/TransactionCommittedTotalCount
invoke Server/myserver/stop

```

The tool can also be used with a graphical MBean explorer. To invoke the Explorer, use the `explore` command in the shell. Figure 20-4 illustrates the Explorer window, showing the MBean structure together with the attributes and values for a selected JMS server.

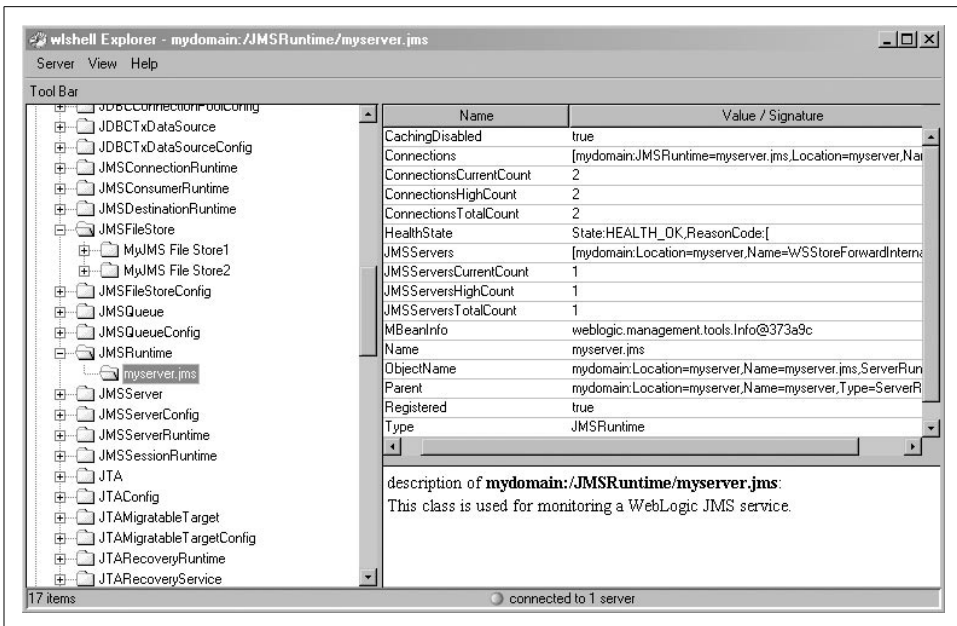


Figure 20-4. WLS Shell's Explorer window

A great feature of the tool is its ability to reverse-engineer a domain's `config.xml` file. This creates a script file that reconstructs the domain's configuration. For any JMX-related work in WebLogic, we recommend that you use this tool because it is far more convenient than the `weblogic.Admin` tool, and its scripting language lets you create complex scripts for monitoring and configuring WebLogic resources.

## Standard MBean Access

A typical JMX client will use the generic methods to interact with an MBean Server and the MBeans it hosts. As we mentioned earlier, an MBeanServer instance is the primary point of contact with an MBean Server. Once you've obtained an Administration or Local home, you can invoke the `getMBeanServer()` method on the MBean home object to retrieve the MBeanServer. Subsequent interaction with the MBeanServer follows the JMX specification.

The following example locates all Log MBean instances and prints out the log filenames:

```
MBeanServer mbs = adminHome.getMBeanServer();
Set s = mbs.queryMBeans(new ObjectName("myClusterDomain:*"), null);
for (Iterator i=s.iterator();i.hasNext();) {
    Object o = i.next();
    ObjectInstance m = (ObjectInstance) o;
    System.err.println(mbs.getAttribute(m.getObjectName(),"FileName"));
}
```

Here, we've used the `queryMBeans()` method to find all MBeans of interest, in this instance all Log MBeans within `myClusterDomain`. Once we've acquired the set of desired MBeans, we can iterate over the collection and use the `getAttribute()` method to read the `FileName` property of each Log MBean. You also can use the `setAttribute()` method to set attributes on the Log MBean, or the `invoke()` method to invoke an operation on the Log MBean.

If your code needs to run on other J2EE platforms, or if it needs to interact with non-WebLogic MBeans, we recommend that you use this approach, which adheres to standard JMX conventions.

## WebLogic MBean Access

WebLogic provides a proprietary, type-safe layer over the standard MBeanServer interface. Instead of obtaining a reference to the MBeanServer instance, you can directly use the MBean home to find or retrieve an MBean. If you know the MBean's name, you can use the `getMBean()` method to retrieve the MBean and cast the returned object to the appropriate type, as specified in WebLogic's API documentation. You then can directly use the methods exposed by WebLogic's MBean interface to query or manipulate the MBean. The following code locates the MBean corresponding to a data source deployed to `ServerA` within `myClusterDomain` and prints out its JNDI name and the name of its associated connection pool:

```
WebLogicObjectName name = new WebLogicObjectName(
    "MyJDBC Data Source","JDBCDataSourceConfig",
    "myClusterDomain", "ServerA");
JDBCDataSourceMBean pf = (JDBCDataSourceMBean) localHomeA.getMBean(name);
System.err.println(pf.getJNDIName());
System.err.println(pf.getPoolName());
```

Notice how the MBean's name is specified using a `WebLogicObjectName` instance. The `JDBCDataSourceMBean` interface (and any WebLogic-specific Configuration or Runtime MBeans) explicitly exposes all the necessary attributes and allowed operations. In general, the MBean's interface will expose `getXXX()` and `setXXX()` methods to read and write each MBean attribute and sensible names for each allowed operation on the target resource or service. Thus, WebLogic-specific MBeans give you a straightforward, type-safe way of exploring MBeans.

If you don't know the MBean's name, you still can query the MBean home for all MBeans using the `getAllMBeans()` method and then operate on the one(s) you want. The following code prints out the JNDI name for all data sources, though it's rather inefficient:

```
Set s = adminHome.getAllMBeans();
for (Iterator i=s.iterator();i.hasNext();) {
    WebLogicMBean o = (WebLogicMBean) i.next();
    if (o instanceof JDBCDataSourceMBean) {
        JDBCDataSourceMBean p = (JDBCDataSourceMBean)o;
        System.err.println(p.getJNDIName());
    }
}
```

Alternatively, you can query the home interface for all MBeans of a particular type directly using the `getMBeansByType()` method and instantly filter out unwanted MBeans. The following code sample shows how to query the MBean home for all data source Configuration MBeans and iterate over the result:

```
Set s = adminHome.getMBeansByType("JDBCDataSourceConfig");
for (Iterator i=s.iterator();i.hasNext();) {
    JDBCDataSourceMBean p = (JDBCDataSourceMBean)i.next();
    System.err.println(p.getJNDIName());
}
```

If your JMX client needs to run on other J2EE platforms or access non-WebLogic MBeans that you have developed and incorporated into WebLogic, you should use the standard MBean access methods and not this WebLogic-specific type-safe approach.

## Examples

WebLogic contains hundreds of MBeans. The following sections cover examples of some of the Runtime, Configuration, and Security MBeans and examines common ways of manipulating these MBeans. Other MBeans provided by WebLogic may be manipulated similarly.

### Runtime MBeans

A prime example of a Runtime MBean is the `ServerRuntimeMBean`, which provides information on the operational status of a server and other details such as its listen

address and port. The following code uses WebLogic's type-safe approach to connect to a Managed Server, print out its listen address and port, and then shut it down:

```
serverRuntime = (ServerRuntimeMBean)
    localHomeB.getRuntimeMBean("ServerB", "ServerRuntime");
System.out.println("Listens on " +
    serverRuntime.getListenAddress()+":"+serverRuntime.getListenPort());
serverRuntime.shutdown();
```

You can do the same thing using the `weblogic.Admin` tool:

```
java weblogic.Admin -url http://serverb.x:7001 -username system -password pssst
    INVOKE -mbean "myClusterDomain:Location=ServerB,Name=ServerB,Type=ServerRuntime"
    -method shutdown
```

The `ServerRuntimeMBean` will exist on the Local Home of each server instance. To find all of the server runtimes, we will have to contact the Administration Server:

```
Set mbeanSet = adminHome.getMBeansByType("ServerRuntime");
Iterator mbeanIterator = mbeanSet.iterator();
while (mbeanIterator.hasNext()) {
    ServerRuntimeMBean serverRuntime = (ServerRuntimeMBean)mbeanIterator.next();
    System.err.println("Found server: " + serverRuntime.getName());
}
```

Runtime MBeans often hold a lot of useful information when it comes to monitoring a resource and its usage. Here, for example, we list the maximum capacity and current connection count for a JDBC pool targeted to a Managed Server, using the standard JMX interface to the `JDBCConnectionPoolRuntimeMBean`:

```
MBeanServer mbs = localHomeB.getMBeanServer();
WebLogicObjectName parent = new WebLogicObjectName("ServerB", "ServerRuntime",
    "myClusterDomain");
WebLogicObjectName oname = new WebLogicObjectName("NoTxPool",
    "JDBCConnectionPoolRuntime", "myClusterDomain", "ServerB", parent);
Set s = mbs.queryMBeans(oname,null);
for (Iterator i = s.iterator(); i.hasNext();) {
    Object o = i.next();
    ObjectInstance m = (ObjectInstance) o;
    System.err.println(mbs.getAttribute(m.getObjectName(), "MaxCapacity"));
    System.err.println(mbs.getAttribute(m.getObjectName(),
        "ActiveConnectionsCurrentCount"));
}
```

In this case, we found the name of the desired MBean by issuing the following command:

```
java weblogic.Admin -url http://10.0.10.10:8001 -username system -password pssst
    GET -pretty -type JDBCConnectionPoolRuntime
```

Similar statistics are available for all kinds of managed resources, including EJBs, JMS servers, web applications, and more.

## Administration MBeans

The Administration MBeans hosted by the Administration Server completely define the configuration of the domain. You can manipulate the Administration MBeans to alter the configuration of the domain. The following example locates a JDBC connection pool and changes its maximum capacity:

```
WebLogicObjectName n = new WebLogicObjectName("MyJDBC Connection Pool",
                                               "JDBCConnectionPool", "myClusterDomain");
JDBCConnectionPoolMBean cp = (JDBCConnectionPoolMBean) adminHome.getMBean(n);
cp.setMaxCapacity(20);
```

If you take a closer look at the JavaDoc documentation, you will find that the `setMaxCapacity()` method on the `JDBCConnectionPoolMBean` is dynamic. This means that the effect of changing the maximum capacity will be immediate, something you can verify for yourself using the Administration Console. In addition, the change is persisted so that the next time the server is started, the new configuration will be in place.

You also can create new Administration MBeans dynamically by using the home methods `createAdminMBean()` or `findOrCreateAdminMBean()`. The latter variant returns an existing MBean if it finds one. There are two main variations of this method call:

```
findOrCreateAdminMBean(String name, String type, String domain)
```

This creates an Administration MBean in the given domain with the appropriate name and type.

```
findOrCreateAdminMBean(String name, String type, String domain,
                       ConfigurationMBean parent)
```

This also creates an Administration MBean with the given name and type, but in addition it makes the newly created MBean a child of the parent.

In the following code sample, we use both variants to create a JMS Connection Factory, Server, and Queue:

```
String domain="myClusterDomain";
JMSConnectionFactoryMBean cf = (JMSConnectionFactoryMBean)
    ahome.findOrCreateAdminMBean("oreillyConnectionFactory",
                                "JMSConnectionFactory", domain);
cf.setJNDIName("oreilly.CF");
JMSServerMBean server = (JMSServerMBean)
    ahome.findOrCreateAdminMBean("oreillyJMSServer",
                                "JMSServer", domain);
JMSDestinationMBean queue = (JMSDestinationMBean)
    home.findOrCreateAdminMBean("oreillyQ", "JMSQueue", domain, server);
queue.setJNDIName("oreilly.Q");
```

In this case, both the `JMSConnectionFactory` and the `JMSServer` MBeans are implicit children of the `DomainMBean`, while the `JMSDestination` MBean associated with the JMS queue is a child of the `JMSServer` MBean.

## Security MBeans

In order to manipulate the Security MBeans, you need to be familiar with WebLogic's SSPI architecture as outlined in Chapter 17. For instance, suppose you need to provide a facility to programmatically add a new WebLogic user to the default security realm. Then, you need to know how to find an Authentication Provider that also implements the optional `UserEditorMBean` interface. Using WebLogic's type-safe interface, the following method shows how to programmatically add new users to WebLogic's default security realm:

```
public void createUsers(MBeanHome ahome, String un, String pw) {
    // First we locate the default security realm for the domain
    RealmMBean securityRealm = ahome.getActiveDomain()
        .getSecurityConfiguration()
        .findDefaultRealm();
    // We then find all the authentication providers
    AuthenticationProviderMBean[] providers =
        securityRealm.getAuthenticationProviders();
    for (int i = 0; i < providers.length; i++) {
        // We look for a provider that implements UserEditorMBean
        if (providers[i] instanceof UserEditorMBean) {
            UserEditorMBean editor = (UserEditorMBean) providers[i];
            try {
                editor.createUser(un, pw, pw);
                System.out.println("Created User " + un);
            } catch (Exception e) {
                System.err.println("Exception " + e.toString());
            }
        }
    }
}
```

It is important to remember that the Security MBeans live in their own Object Name domain called `Security`. For instance, the name of the default Authenticator MBean is `Security:Name=myrealmDefaultAuthenticator`. Using this information, we can quite easily use the Administration Tool to create a new WebLogic user from the command line:

```
java weblogic.Admin -username system -password pssst -url http://10.0.10.10:7001
    INVOKE -mbean "Security:Name=myrealmDefaultAuthenticator"
        -method createUser username password password
```

## MBean Notifications

The Administration, Local Configuration, Security, and Runtime MBeans allow us to view and modify the configuration and runtime statistics of resources being managed by the MBean Server. Notifications and Monitoring go one step further—they allow you to react to changes in the runtime state of the underlying managed resources. This is critical if you want to build a management application that is able to react and respond to runtime state changes of WebLogic resources and services.

The JMX specification defines a model that allows MBeans to broadcast management events called notifications. You then can create applications that listen for JMX notifications. Your applications may also filter out unwanted notifications. Each WebLogic MBean directly or indirectly extends the `javax.management.NotificationBroadcaster` interface, thereby allowing you to add or remove notification listeners.

## Creating a Notification Listener

A *notification listener* is a handler that is triggered when the MBean it is registered with sends one or more JMX notifications. A notification listener is an instance of a class that implements the `javax.management.NotificationListener` interface. For remote applications, the listener object should instead implement the `weblogic.management.RemoteNotificationListener` interface, which merely extends the previous interface and `java.rmi.Remote` as well, making the JMX notifications available to external clients via RMI.

The listener interface is simple—it exposes a single method, `handleNotification()`, that gets called when a notification is received. Here is an example of a remote notification listener:

```
public class SimpleListener implements RemoteNotificationListener {
    public void handleNotification(Notification notification, Object hb) {
        System.err.println("Received Notification");
        System.err.println("Source=" + notification.getSource());
        System.err.println("Message=" + notification.getMessage());
        System.err.println("Type=" + notification.getType());
    }
}
```

## Registering a Listener

In order to receive notifications from an MBean, you need to register a listener object with the MBean. If you already hold a reference to the MBean, you simply can invoke the `addNotificationListener()` method to register the listener object. An alternative is to use the `addNotificationListener()` method on the `MBeanServer`, which takes the name of the MBean and the listener object and registers the listener for you. The following code registers a listener with the `JDBCConnectionPoolMBean` called `My Connection Pool`:

```
SimpleListener sl = new SimpleListener();
WebLogicObjectName oname = new WebLogicObjectName("My Connection Pool",
    "JDBCConnectionPool", "myClusterDomain");
mb.getMBeanServer().addNotificationListener(oname, sl, null, null);
```

If we then change the runtime state of the JDBC connection pool (e.g., modify the initial capacity of the pool), and if the program that registered the listener object is still running, the listener will receive a notification of the change. Here is the output

that is generated when the initial capacity was raised from 2 to 3 using the Administration Console:

```
Received Notification
Source=myClusterDomain:Name=My Connection Pool,Type=JDBCConnectionPool
Message=WebLogic MBean Attribute change for InitialCapacity from 2 to 3
Type=jmx.attribute.change
```

The MBean fires notifications whenever attributes are modified, or whenever new attributes are added or existing ones are removed from an MBean. WebLogic provides two subclasses of the standard Notification class that represent notification events that are fired when attributes are added or removed from an MBean:

`weblogic.management.AttributeAddNotification`

An instance of this class is fired whenever an `addXXX()` method is called on an MBean.

`weblogic.management.AttributeRemoveNotification`

An instance of this class is fired whenever a `removeXXX()` method is called on an MBean.

Notifications also are used by the Monitor MBeans that we explore in the next section, and by the distributed logging framework covered in Chapter 21.

## Monitor MBeans

To further enhance your control over the management of an application, JMX provides Monitor MBeans that allow you to observe attribute values as they vary over time and to fire notifications when these values exceed specific thresholds.

Monitor MBeans monitor attributes in other MBeans at specified intervals and derive a value from this observation called the *derived gauge*. The derived gauge is either the exact value of the attribute, or optionally the difference between two consecutive observed values of a numeric attribute. Depending on the Monitor MBean and its setup, it then can emit an MBean Notification. Monitors can also send notifications when error cases are encountered during monitoring.

Monitors are MBeans as well, and so can be created or destroyed dynamically. Typically, Monitor MBeans are used in combination with the statistics captured by the Runtime MBeans. For example, you could write an MBean to monitor the connection delay time of connections in a JDBC connection pool, or the number of messages dropped in a JMS destination. The Monitor MBean could then send a notification when certain thresholds are crossed. This gives you the opportunity to listen for those notifications and take some appropriate action.

## Types of Monitors

There are three types of Monitor MBeans, each explained in depth in the JMX specification:

### CounterMonitor

This can observe Integer attributes that behave like a counter. That is, the attribute value is always greater than or equal to zero, they can only be incremented, and they may roll over. This monitor sends a notification when the derived gauge exceeds a threshold, after which you can have the monitor automatically increase the threshold by some offset.

### GaugeMonitor

This can observe Integer, Float, or Double attributes that behave like gauges, arbitrarily increasing or decreasing. Notifications are sent when values exceed a high or low threshold. A hysteresis mechanism ensures that repeated triggering of notifications doesn't occur.

### StringMonitor

This can observe String attributes. The derived gauge is always the value of the attribute, and the monitor fires events when the observed attribute differs from some initialized String value.

Notifications sent by these monitors are all instances of the `MonitorNotification` class. This subclass of the usual `Notification` event class includes information such as the observed MBean's object name, attribute name, derived gauge, and threshold value or string that triggered the notification. Note that the `type` property of the `Notification` class is a string that represents the type of monitor notification. This notification type is a string of the form `jmx.monitor.*`. For example, the Gauge monitor will send notifications of two possible types: `jmx.monitor.gauge.low` or `jmx.monitor.gauge.high`.

## An Example Monitor

As mentioned earlier, the JMX specification provides an in-depth description of these standard JMX monitors. Here, we'll look at how to create a Monitor MBean for WebLogic. This example creates a Counter Monitor MBean that observes the invocation count on the `FileServlet`, which is responsible for serving requests for static files within a particular web application.

The first thing we want to do is create a new `NotificationListener` specialized to handle Monitor notifications so that we can access the additional information available:

```
public class MyMonitorListener implements RemoteNotificationListener {
    public void handleNotification(Notification notification, Object obj) {
        System.err.println("Received Notification");
        System.err.println("Message: " + notification.getMessage());
        System.err.println("Type: " + notification.getType());
        if (notification instanceof MonitorNotification) {
            MonitorNotification monitorNotification =
                (MonitorNotification) notification;
            System.out.println("This is a MonitorNotification");
        }
    }
}
```

```

        System.out.println("Observed Attribute: "
            + monitorNotification.getObservedAttribute());
        System.out.println("Observed Object: "
            + monitorNotification.getObservedObject());
        System.out.println("Trigger value: " + monitorNotification.getTrigger());
    }
}
}

```

When you monitor something, you need to construct `ObjectName`s for both the Monitor MBean and the target MBean that needs to be monitored. Here, we've created names for both the Counter Monitor MBean and FileServlet MBean:

```

WebLogicObjectName monitorObjectName = new
    WebLogicObjectName("myClusterDomain:Type=CounterMonitor,Name=OurCounter");

WebLogicObjectName myServlet = new
    WebLogicObjectName("myClusterDomain:Location=ServerB,Name=ServerB_ServerB_" +
        "DefaultWebApp_weblogic.servlet.FileServlet_94," +
        "ServerRuntime=ServerB,Type=ServletRuntime");

```

In this case, the FileServlet MBean is deployed under the DefaultWebApp on ServerB. Its exact name may differ depending on your deployment.

Our Counter Monitor needs to observe the `InvocationTotalCount` property, whose value gets incremented every time the DefaultWebApp serves up a static file. We now can create a CounterMonitor and configure it to send a notification only if more than 15 requests have been made, and subsequently with an offset value of 10:

```

CounterMonitor monitor = new CounterMonitor();
monitor.setObservedAttribute("InvocationTotalCount");
monitor.setThreshold(new Integer(15));
monitor.setOffset(new Integer(10));
monitor.setNotify(true);
monitor.setObservedObject(myServlet);

```

Finally, we need to instantiate our listener object, bind it to the monitor, register our monitor with the jmx, and start the monitor:

```

MyMonitorListener listener = new MyMonitorListener();
monitor.addNotificationListener(listener, null, null);
monitor.preRegister(mb.getMBeanServer(), monitorObjectName);
monitor.start();

```

After requesting a static file (say, `index.html` within the web application) a number of times, the code produces the following output:

```

Received Notification
Message:
Type: jmx.monitor.counter.threshold
This is a MonitorNotification
Observed Attribute: InvocationTotalCount
Observed Object: myClusterDomain:Location=ServerB,Name=ServerB_ServerB_DefaultWebApp_
weblogic.
    servlet.FileServlet_94,ServerRuntime=ServerB,Type=ServletRuntime
Trigger value: 15

```

In fact, we observe that further notifications are fired when the same static file receives 25 hits, again when it receives 35 hits, and so on.

## Timer MBeans

JMX provides a standard timer service API, which can be used to generate notifications at set times or intervals. WebLogic 8.1 implements this service by extending the standard JMX timer service, enabling it to run with WebLogic execute threads and any associated security context.

To use this service, you must use an instance of the `weblogic.management.timer.Timer` class. The following example illustrates how to create an instance, register a listener, register a notification for when an event should be emitted, and start the timer:

```
Timer timer= new Timer();
// Register a standard notification listener
timer.addNotificationListener(someListener,
                             null, "handback object");
// Start in one second
Date start =
    new Date((new Date()).getTime() + 1000L);
// Repeat every minute, please
notificationId = timer.addNotification("eggTimer",
                                       "someString", this, start, 3*Timer.ONE_MINUTE
);
// Start the timer
timer.start();
```

Note that you may register any standard notification listener that we have already encountered. The type of the notification is `TimerNotification`. The only tricky bit is the call to the `addNotification()` method, which lets you set up the timer schedule. You can invoke this method as many times as you like to add multiple schedules to the timer. One of the method signatures looks like this:

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

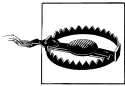
Other versions of the same method let you omit the period and/or the number of occurrences. Let's take a closer look at the arguments of this method:

- Use the `type` argument to identify the type of notification.
- Use the `message` argument to set the string that should appear in the `message` attribute of the `TimerNotification`.
- Use the `userData` argument to pass an object to the listeners. This can be anything that a listener needs to be able to access. In our example, we simply used `this`.
- Use the `startTime` argument to set the time and date after which notifications should start being sent.

- If you specify a period (.), this argument determines the number of milliseconds between notifications. Repeat notifications are disabled if you set this argument to 0.
- Use the `numOccurrences` argument to set the total number of times that the notification is fired. The MBean keeps track of the number of notifications that are yet to be fired, and you can invoke the `Timer.getNbOccurrences()` method to retrieve this information. If this value is set to 0 and you have specified a period, the notification will repeat indefinitely.

The `Timer` class also defines constants that make it easier for you to specify the time period: `ONE_SECOND`, `ONE_MINUTE`, `ONE_HOUR`, `ONE_DAY`, `ONE_WEEK`. For example, `ONE_WEEK` resolves to the number of milliseconds in a week.

`hod`, which simply removes all notifications assigned to a timer.



Timers are not persistent. If you reboot your server, all timers will be lost and you will have to reinitiate them.

The `addNotification()` method returns an `Integer` identifier that can be used later to remove the notification from the timer. For example, you could write the following:

```
// Later
timer.stop();
timer.removeNotification(notificationId);
//alternatively
timer.removeNotification("eggTimer");
```

Alternatively, you may use the `removeNotification(type)` method to remove all notifications of the given type, or else you can use the `removeAllNotifications()` method, which simply removes all notifications assigned to a timer.