BEA WebLogic Server®

Developing Manageable Applications with JMX

Version 9.0
Revised: September 7, 2005

Copyright

Copyright © 2005 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

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

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

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

Trademarks or Service Marks

BEA, BEA JRockit, BEA Liquid Data for WebLogic, BEA WebLogic Server, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic JRockit, BEA WebLogic Log Central, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic WorkGroup Edition, BEA WebLogic Workshop, and Liquid Computing are trademarks of BEA Systems, Inc. BEA Mission Critical Support is a service mark of BEA Systems, Inc. All other company and product names may be the subject of intellectual property rights reserved by third parties.

All other trademarks are the property of their respective companies.

Contents

Introduction and Roadmap

Document Scope and Audience 1-2
Guide to this Document 1-2
Related Documentation 1-2
Samples for the JMX Developer 1-3
Avitek Medical Records Application (MedRec) and Tutorials 1-3
New and Changed JMX Features in This Release 1-4
JMX 1.2 and JMX Remote API 1.0 (JSR-160) 1-4
Deprecated MBeanHome and Type-Safe Interfaces 1-4
Changes to the Model for Distributing Configuration Data in a Domain 1-5
Changes to the MBean Data Model 1-6
New Functionally Aligned MBean Servers 1-7
Facilities for Registering Custom MBeans 1-8
New Reference Document for WebLogic Server MBeans 1-8
. . . . 1-8

Understanding JMX

What Management Services Can I Develop with JMX? 2-1
Advanced JMX-Programming: Creating Management-Aware Applications . . . 2-2
When Is It Appropriate to Use JMX? 2-2
What Management Services Have BEA Partners Developed? 2-3
Anatomy of JMX 2-3

JMX Layers 2-4
Indirection and Introspection 2-4
Notifications and Monitor MBeans 2-5
How JMX Notifications are Broadcast and Received 2-6
Active Polling with Monitor MBeans 2-7

Designing Manageable Applications

Best Practices 3-1
Use Standard MBeans 3-2
Use the Runtime MBean Server 3-2
Using the JVM Platform MBean Server 3-3
Use ApplicationLifecycleListener to Register Application MBeans 3-4
Alternatives That Use Only JDK Classes 3-4
Unregister Application MBeans When Applications Are Undeployed 3-5
For EJBs and Servlets Place Management Logic in a Delegate Class 3-5
Use Open MBean Data Types 3-7
Emit Notifications Only When Necessary 3-7
Additional Design Considerations 3-7

Instrumenting and Registering Custom MBeans

Create and Implement a Management Interface 4-3
Modify Business Methods to Push Data 4-6
Register the MBean 4-7
Package Application and MBean Classes 4-9

Using the WebLogic Server JMX Timer Service

Using the WebLogic Timer Service: Main Steps 5-2
Configuring a Timer MBean to Emit Notifications 5-2
Creating Date Objects 5-4

Example: Generating a Notification Every Five Minutes After 9 AM 5-5
Removing Notifications. 5-8

Accessing Custom MBeans

Accessing Custom MBeans from JConsole. 6-1
Accessing Custom MBeans from WebLogic Scripting Tool 6-2
Accessing Custom MBeans from an Administration Console Extension 6-2

Index

CHAPTER 1

Introduction and Roadmap

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

This document describes using JMX to make your applications manageable.

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

- "Document Scope and Audience" on page 1-2
- "Guide to this Document" on page 1-2
- "Related Documentation" on page 1-2
- "Samples for the JMX Developer" on page 1-3
- "New and Changed JMX Features in This Release" on page 1-4

Document Scope and Audience

This document is a resource for software developers who develop management services for J2EE applications. 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.

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

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 JMX,” gives an overview of JMX and describes how J2EE applications can use JMX.
- Chapter 3, “Designing Manageable Applications,” recommends design patterns for making J2EE applications manageable through JMX.
- Chapter 4, “Instrumenting and Registering Custom MBeans,” describes how to create your own MBeans (custom MBeans), which enable you to promote your application to the status of a managed object within a larger management system.
- Chapter 5, “Using the WebLogic Server JMX Timer Service,” describes how to configure your JMX client to carry out a task at a specified time or a regular time interval by using WebLogic Server’s implementation of the JMX timer service.
- Chapter 6, “Accessing Custom MBeans,” describes options for accessing your MBeans (other than through JMX).

Related Documentation

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

To view the JMX 1.2 specification, 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 *Developing Applications with WebLogic Server*.

Samples for the JMX Developer

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

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

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from

the `WL_HOME/samples/domains/medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

MedRec has instrumented one of its session EJBs for management through JMX. The EJB keeps track of how many times it writes to the database, and exposes this counter as an MBean attribute.

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)" on page 1-4
- "Deprecated MBeanHome and Type-Safe Interfaces" on page 1-4
- "Changes to the Model for Distributing Configuration Data in a Domain" on page 1-5
- "Changes to the MBean Data Model" on page 1-6
- "Facilities for Registering Custom MBeans" on page 1-8
- "New Functionally Aligned MBean Servers" on page 1-7
- "Facilities for Registering Custom MBeans" on page 1-8
- "New Reference Document for WebLogic Server MBeans" on page 1-8

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

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

Also in 9.0, 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.

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

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

WebLogic Server 9.0 enables you to collect modifications to a domain's configuration and distribute them as a group throughout the domain. The 9.0 release also contains APIs that you can use to monitor the distribution of changes.

In 9.0, 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 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 without restarting a server instance and the replica and its master administration MBean would contain different values. In addition, JMX clients could directly access local configuration MBeans and change their values, which also resulted in an inconsistent state between replica and master MBean.

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 9.0:

- There were inconsistencies in the data model 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 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 `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`). The parent also contains a `lookup`*`Child`* 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 when they want 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*.

Facilities for Registering Custom MBeans

Prior to 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

Understanding JMX

Java Management Extensions (JMX) is a specification for monitoring and managing Java applications. It enables a generic management system to monitor your application, raise notifications when the application needs attention, and change the state of your application to remedy problems. Like SNMP and other management standards, JMX is a public specification and many vendors of commonly used monitoring products support it.

WebLogic Server uses the Java Management Extensions (JMX) 1.2 implementation that is included in JDK 1.5. The following sections describe how Java applications can use JMX to expose runtime metrics and control points to management systems:

- "What Management Services Can I Develop with JMX?" on page 2-1
- "When Is It Appropriate to Use JMX?" on page 2-2
- "What Management Services Have BEA Partners Developed?" on page 2-3
- "Anatomy of JMX" on page 2-3

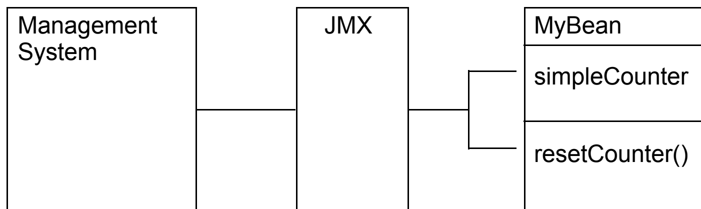
For information about other APIs and utilities that you can use to manage J2EE applications on WebLogic Server, refer to "Overview of WebLogic Server System Administration" in *Introduction to WebLogic Server*.

What Management Services Can I Develop with JMX?

When used to monitor and manage applications, JMX typically provides management applications access to properties in your Java classes that collect management data (see Figure 2-1). Often, these class properties are simple counters that keep track of the resources your

application is consuming. JMX can also provide access to methods in your Java classes that start or stop processes in the application or reset the value of the class properties. Any class that exposes management data through JMX is called a managed bean (MBean). Class properties that are exposed through MBeans are called attributes and methods that are exposed through MBeans are called operations.

Figure 2-1 JMX Provides Access to Management Properties



Once you provide this type of access to JMX-enabled management utilities, system administrators or the operations staff can integrate the data into their overall view of the system. They can use a JMX management utility to view the current value of an MBean attribute, or they can set up JMX monitors to periodically poll the value of your MBean attributes and emit notifications to the management utility only when the values exceed specific thresholds.

Advanced JMX-Programming: Creating Management-Aware Applications

Instead of placing all management responsibility on system administrators or the operations staff, you can create management-aware applications that monitor MBeans and then perform some automated task. For example:

- An application that monitors connection pools and grows or shrinks the pools to meet demand.
- A portal application that monitors the set of deployed applications. If a new application is deployed, the portal application automatically displays it as a new portlet.
- An application that listens for deployments of connector modules and then configures itself to use newly deployed modules.

When Is It Appropriate to Use JMX?

Any critical J2EE application that is a heavy consumer of resources, such as database or JMS connections or caches, should provide some facility for monitoring the application's resource consumption. For these kinds of applications, which might be writing or reading from a database many times each minute, it isn't feasible to use logging facilities to output messages with each

write and read operation. Instead, using JMX for this type of monitoring enables you to write management (instrumentation) code that is easy to maintain and that optimizes your use of network resources.

If you want to monitor basic runtime metrics for your application, WebLogic Server already provides a significant number of its own MBeans that you can use (see Best Practices: Listening for WebLogic Server Events in *Developing Custom Management Utilities with JMX*). For example, you can use existing WebLogic Server MBeans to track the hit rate on your application's servlets and the amount of time it takes to process servlet requests.

Although WebLogic Server MBeans can indicate to an operations center the general state of its resources, it cannot provide detailed information about how a specific application is using the resources. For example, WebLogic Server MBeans can indicate how many connections are being used in a connection pool, but they do not indicate which applications are using the connection pools. If your domain contains several active applications and you notice that some connections are always in use, consider creating MBeans that monitor when each application session gets and releases a connection. You could also include a management operation that ends sessions that appear to be stuck.

In addition, if your application creates and maintains its own cache or writes to a data repository that is outside the control of the application container, consider creating MBeans to monitor the size of the cache or the amount of data written to the repository.

What Management Services Have BEA Partners Developed?

BEA Partners have developed an extensive set of management consoles that can monitor and analyze data from WebLogic Server MBeans and potentially from MBeans that you develop for your own applications. These consoles can integrate WebLogic Server into an overall management strategy for your network or data center operations. To see the list of management software available, visit the Partners page on www.bea.com.

Anatomy of JMX

The following sections describe JMX:

- "JMX Layers" on page 2-4
- "Indirection and Introspection" on page 2-4
- "Notifications and Monitor MBeans" on page 2-5

JMX Layers

Like most of J2EE, JMX is a component-based technology in which different types of software vendors provide different types of components. This division of labor enables each type of vendor to focus on providing only the software that falls within its area of expertise. JMX organizes its components into the following layers:

- Instrumentation

  Applications that you write, resources, and other manageable objects provide this layer. In this layer, application developers create JMX components called managed beans (MBeans), which contain the properties (attributes) and methods (operations) that they want to expose to external management systems.

- Agent

  The JVM and application servers such as WebLogic Server provide this layer. This layer includes a registry of MBeans and standard interfaces for creating, destroying, and accessing MBeans.

  The agent layer also provides services for remote clients as well as a monitoring and a timer service. For more information, see "Using the WebLogic Server JMX Timer Service" on page 5-1 and Using Notifications and Monitor MBeans in *Developing Custom Management Utilities with JMX*.

- Distributed Services

  Management consoles or other J2EE applications provide this layer. In this layer, a management application sends or receives requests from the agent. Often, this layer is available as a plugin or adapter that enables a management console to support a variety of management protocols, such as JMX and SNMP.

Indirection and Introspection

Two key concepts for understanding JMX are indirection and introspection, which enable a JMX application to manage proprietary resources without needing access to proprietary class definitions.

The general model for JMX is that applications in the distributed services layer never interact directly with classes in the instrumentation layer. Instead, under this model of indirection, the JMX Agent provides standard interfaces, such as `javax.management.MBeanServerConnection`, that:

- Expose a class's management interface to management clients in the distributed services layer
- Receive requests from management clients, such as a request to get the value of a property that a class is exposing through JMX
- Interact with the class to carry out the request and returns the result to the management client

Each class describes to the MBean server the set of properties and methods that it wants to expose through JMX. A property that a class exposes through JMX is called an MBean **attribute**, and a method that it exposes is called an **operation**. JMX specifies multiple techniques (design patterns) that a class can use to describe its attributes and operations, and these design patterns are formalized as the following MBean types: standard, dynamic, model, and open.

A class that instruments the standard MBean type describes its management interface in way that is most like Java programming: a developer creates a JMX interface file that contains getter and setter methods for each class property that is to be exposed through JMX. The interface file also contains a wrapper method for each class method that is to be exposed. Then the class declares the name of its JMX interface. When you register a standard MBean with the MBean server, the MBean server introspects the class and its JMX interface to determine which attributes and operations it will expose to the distributed services layer. The MBean server also creates an object, `MBeanInfo`, that describes the interface. Management clients inspect this `MBeanInfo` object to learn about a class's management interface.

A class that instruments the model MBean type describes its management interface by constructing its own `MBeanInfo` object, which is a collection of metadata objects that describe the properties and methods to expose through JMX. When you register a model MBean with the MBean server, the MBean server uses the existing `MBeanInfo` object instead of intropsecting the class.

Notifications and Monitor MBeans

JMX provides two ways to monitor changes in MBeans: MBeans can emit notifications when specific events occur (such as a change in an attribute value), or monitor MBeans can poll an MBean periodically to retrieve the value of an attribute.

The following sections describe JMX notifications and monitor MBeans:

- "How JMX Notifications are Broadcast and Received" on page 2-6
- "Active Polling with Monitor MBeans" on page 2-7

How JMX Notifications are Broadcast and Received

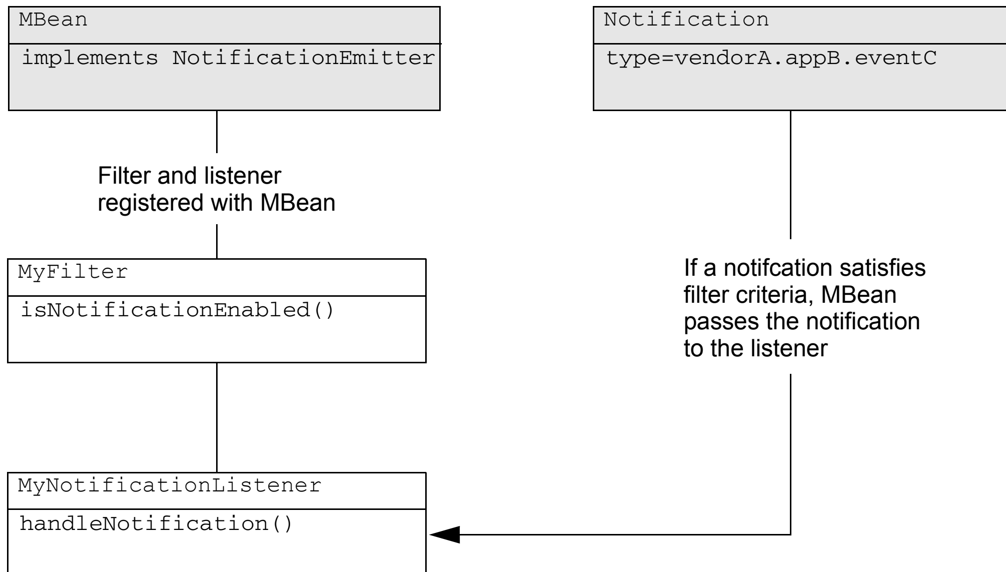
As part of creating an MBean, you can implement the `javax.management.NotificationEmitter` interface, which enables the MBean to emit notifications when different types of events occur. For example, you create an MBean that manages your application's use of a connection pool. You can configure the MBean to emit a notification when the application creates a connection and another notification when the application drops a connection.

To listen for notifications, you create a listener class that implements the `javax.management.NotificationListener.handleNotification()` method. Your implementation of this method includes the logic that causes the listener to carry out an action when it receives a notification. After you create the listener class, you create another class that registers the listener with an MBean.

By default, an MBean broadcasts all of its notifications to all of its registered listeners. However, you can create and register a filter for a listener. A filter is a class that implements the `javax.management.NotificationFilter.isNotificationEnabled()` method. The implementation of this method specifies one or more notification types. (In this case, **type** refers to a unique string within a notification object that identifies an event, such as `vendorA.appB.eventC`.) When an event causes an MBean to generate a notification, the MBean invokes a filter's `isNotificationEnabled()` method before it sends the notification to the listener. If the notification type matches one of the types specified in `isNotificationEnabled()`, then the filter returns `true` and the MBean broadcasts the message to the associated listener.

Figure 2-2 shows a basic system in which a notification listener receives only a subset of the notifications that an MBean broadcasts.

Figure 2-2 Receiving Notifications from an MBean



For a complete description of JMX notifications, refer to the JMX 1.2 specification. See "Related Documentation" on page 1-2.

Active Polling with Monitor MBeans

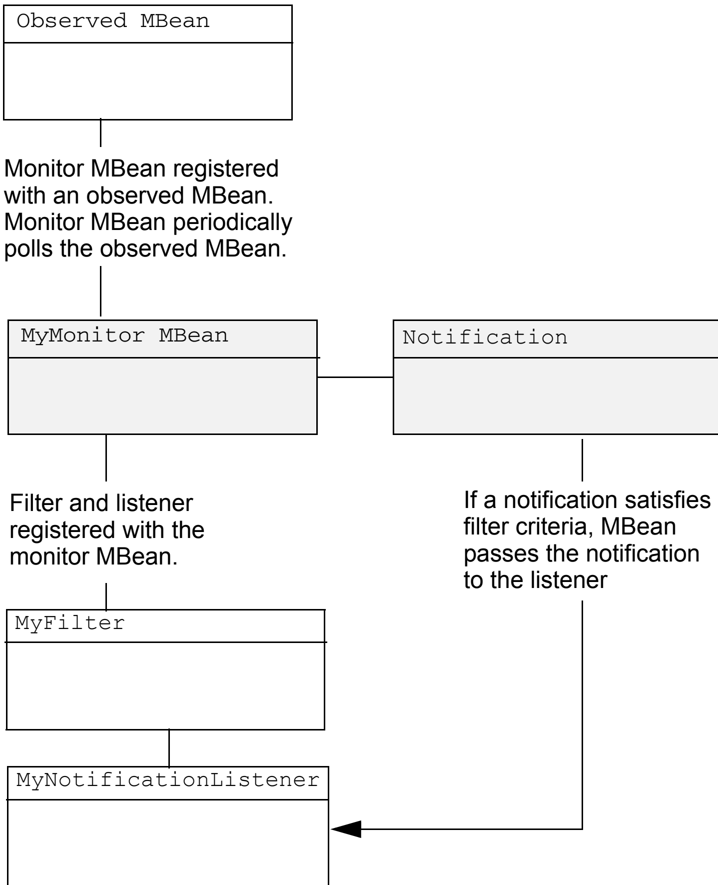
JMX includes specifications for a type of MBeans called **monitor MBeans**, which can be instantiated and configured to periodically observe other MBeans. Monitor MBeans emit JMX notifications only if a specific MBean attribute has changed beyond a specific threshold. A monitor MBean can observe the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

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

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

Figure 2-3 shows a basic system in which a monitor MBean is registered with an MBean. A `NotificationListener` is registered with the monitor MBean, and it receives notifications when the conditions within the monitor MBean are satisfied.

Figure 2-3 Monitor MBeans



CHAPTER 3

Designing Manageable Applications

Within JMX, there are several viable design patterns and deployment options that you can use to make your application manageable. BEA's recommended design patterns assume that the instrumentation of your Java classes should:

- Use as few system resources as possible; management functions must not interfere with business functions.
- Be separate from your business code whenever possible.
- Deploy along with the business code and share its life cycle; you should not require the operations staff to take additional steps to enable the management of your application.

The following sections describe designing manageable applications:

- "Best Practices" on page 3-1
- "Additional Design Considerations" on page 3-7

Best Practices

The following sections describe BEA's recommendations for designing manageable applications:

- "Use Standard MBeans" on page 3-2
- "Use the Runtime MBean Server" on page 3-2
- "Use ApplicationLifecycleListener to Register Application MBeans" on page 3-4

- "Unregister Application MBeans When Applications Are Undeployed" on page 3-5
- "For EJBs and Servlets Place Management Logic in a Delegate Class" on page 3-5
- "Use Open MBean Data Types" on page 3-7
- "Emit Notifications Only When Necessary" on page 3-7

Use Standard MBeans

Of the many design patterns that JMX defines, BEA recommends that you use standard MBeans, which are the easiest to code. In the simplest design pattern for standard MBeans, you do the following:

1. Create an interface for the management properties and operations that you want to expose.
2. Implement the interface in your Java class.
3. Invoke the `javax.management.MBeanServerConnection.createMBean()` method and pass your management interface in the method's parameter.

The MBean server introspects your interface, finds the implementation, and registers the interface and implementation as an MBean.

In this design pattern, the management interface and its implementation must follow strict naming conventions so that the MBean server can introspect your interface. You can circumvent the naming requirements by having your Java class extend `javax.management.StandardMBean`. See `StandardMBean` in the *J2SE 5.0 API Specification*.

Use the Runtime MBean Server

A JVM can contain multiple MBean servers, and another significant design decision is whether to register your MBeans in the JVM's platform MBean server or the WebLogic Server Runtime MBean Server.

As of JDK 1.5, processes within a JVM (local processes) can instantiate a platform MBean server, which is provided by the JDK and contains MBeans for monitoring the JVM itself. Local classes can also register MBeans in this MBean server.

In addition to the platform MBean server, the JVM for any WebLogic Server instance also contains a Runtime MBean Server. (The Administration Server also contains a Domain Runtime MBean Server and an Edit MBean Server, but the Runtime MBean Server is the only one that allows the registration of custom MBeans. See MBean Servers in *Developing Custom Management Utilities with JMX*).

BEA recommends that you register custom MBeans in its Runtime MBean Server. With this option:

- Your MBeans exist in the same MBean server as WebLogic Server MBeans. Remote JMX clients need to maintain only a single connection to monitor your application's MBeans and WebLogic Server MBeans.
- JMX clients must authenticate and be authorized through the WebLogic Server security framework to access your custom MBeans and WebLogic Server MBeans.

The Runtime MBean Server registers its `javax.management.MBeanServer` interface in the JNDI tree. See Make Local Connections to the Runtime MBean Server in *Developing Custom Management Utilities with JMX*.

Using the JVM Platform MBean Server

If it is essential that JMX clients be able to monitor your custom MBeans, WebLogic Server MBeans, and the JVM's platform MBeans through a single MBean server, then you can configure the Runtime MBean Server to be the platform MBean server. With this option:

- Local applications can access all of the MBeans through the `MBeanServer` interface that `java.lang.management.ManagementFactory.getPlatformMBeanServer()` returns.

  Warning: With this local access, there are no WebLogic Server security checks to make sure that only authorized users can access WebLogic Server MBeans. Any application that is running in the JVM can access any of the WebLogic Server MBeans in the Runtime MBean Server/JDK platform MBean Server. **Do not use this configuration if you cannot control or cannot trust the applications that are running within a JVM**.

- If you want to enable remote JMX clients to access custom MBeans, JMX MBeans, and WebLogic Server MBeans, consider the following configuration:
  - The WebLogic Server Runtime MBean Server is configured to be the platform MBean server.
  - Remote access to the platform MBean server is not enabled.

    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.

– Remote JMX clients access JVM MBeans by connecting to the Runtime MBean Server.

To configure the WebLogic Runtime MBean Server to be the JDK platform MBean Server, set the WebLogic `JMXMBean PlatformMBeanServerEnabled` attribute to true and restart the servers in the domain. See `JMXMBean` in the *WebLogic Server MBean Reference*.

Use ApplicationLifecycleListener to Register Application MBeans

If you are creating MBeans for EJBs, servlets within Web Applications, or other modules that are deployed, and if you want your MBeans to be available as soon as you deploy your application, listen for notifications from the deployment service. When you deploy an application (and when you start a server on which you have already deployed an application), the WebLogic Server deployment service emits notifications at specific stages of the deployment process. When you receive a notification that the application has been deployed, you can create and register your MBeans.

There are two steps for listening to deployment notifications with `ApplicationLifecycleListener`:

1. Create a class that extends `weblogic.application.ApplicationLifecycleListener`. Then implement the `ApplicationLifecycleListener.postStart` method to create and register your MBean in the MBean server. The class will invoke your `postStart()` method only after it receives a `postStart` notification from the deployment service. See Programming Application Lifecycle Events in *Developing Applications with WebLogic Server*.

2. In the `weblogic-application.xml` deployment descriptor, register your class as an application listener class.

For an example of this technique, see the MedRec example server.

Alternatives That Use Only JDK Classes

Using BEA's `ApplicationLifecycleListener` is the easiest technique for making an MBean share the life cycle of its parent application. If you do not want to use proprietary WebLogic Server classes and deployment descriptor elements for managing a servlet or an EJB, you can do the following:

- For a servlet, configure a `javax.servlet.Filter` that creates and registers your MBean when a servlet calls a specific method or when the servlet itself is instantiated. See `Filter` in the *J2SE 5.0 API Specification*.
- For an EJB, implement its `javax.ejb.EntityBean.ejbActivate()` method to create and register your MBean. For a session EJB whose instances share a single MBean instance, include logic that creates and registers your MBean only if it does not already exist. See `EntityBean` in the *J2SE 5.0 API Specification*.

Unregister Application MBeans When Applications Are Undeployed

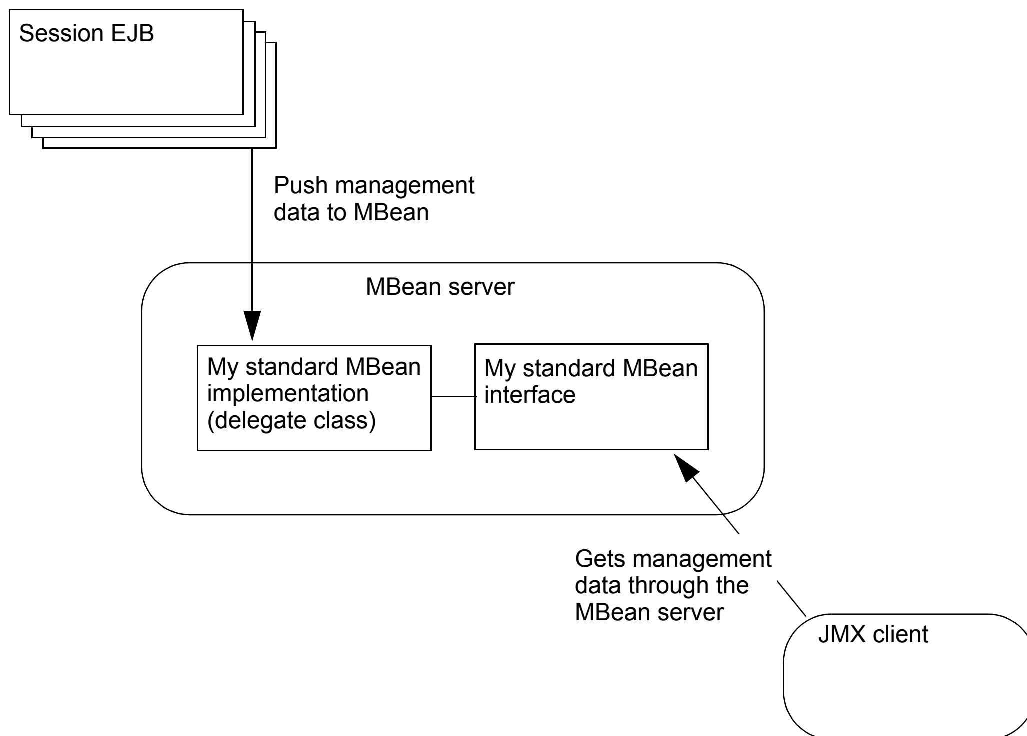
Regardless of how you create your MBeans, BEA recommends that you unregister your MBeans whenever you receive a deployment notification that your application has been undeployed. Failure to do so introduces a potential memory leak.

If you create a class that extends `ApplicationLifecycleListener`, you can implement the `ApplicationLifecycleListener.preStop` method to unregister your MBeans.

For EJBs and Servlets Place Management Logic in a Delegate Class

If you want to expose management attributes or operations for any type of EJB (session, entity, message) or servlet, BEA recommends that you implement the management attributes and operations in a separate, delegate class so that your EJB or servlet implementation classes contain only business logic, and so that their business interfaces present only business logic. See Figure 3-1.

Figure 3-1 Place Management Properties and Operations in a Delegate Class



In Figure 3-1, business methods in the EJB push their data to the delegate class. For example, each time a specific business method is invoked, the method increments a counter in the delegate class, and the MBean interface exposes the counter value as an attribute. For an example of this technique, see the MedRec example server.

This separation of business logic from management logic might be less efficient than combining the logic into the same class, especially if the counter in the delegate class is incremented frequently. However, in practice, most JVMs can optimize the method calls so that the potential inefficiency is negligible.

If this negligible difference is not acceptable for your application, your business class in the EJB can contain the management value and the delegate class can retrieve the value whenever a JMX client requests it.

Use Open MBean Data Types

If remote JMX client will access your custom MBeans, BEA recommends that you limit the data types of your MBean attributes and the data types that your operations return to those defined in `javax.management.openmbean.OpenType`. All JVMs have access to these basic types. See `OpenType` in the *J2SE 5.0 API Specification*.

If your MBeans expose other data types, the types must be serializable and the remote JMX clients must include your types on their class paths.

Emit Notifications Only When Necessary

Each time an MBean emits a notification, it uses memory and network resources. For MBean attributes whose values change frequently, such memory and resource uses might be unacceptable.

Instead of configuring your MBeans to emit notifications each time its attributes change, BEA recommends that you use monitor MBeans to periodically poll your custom MBeans to determine whether attributes have changed. You can configure the monitor MBean to emit a notification only after an attribute changes in a specific way or reaches a specific threshold.

For more information, see Best Practices: Listening Directly Compared to Monitoring in *Developing Custom Management Utilities with JMX*.

Additional Design Considerations

In addition to BEA's best practices, consider the following:

- While you might design one managed object for each business object, there is no requirement for how your management objects should relate to your business objects. One management object could aggregate information from multiple business objects or conversely, you could split information from one business object into multiple managed objects.

  For example, if a servlet uses multiple helper classes and you want one MBean to represent the servlet, each helper class push its management data into a single MBean implementation class.

  The organization that you choose depends on the number of MBeans you want to provide to the system administrator or operations staff contrasted with the difficulty of maintaining a complex management architecture.

- If you package your management classes in an application's `APP-INF` directory, all other classes in the application can access them. If you package the classes in a module's archive file, then only the module can access the management classes.

  For example, consider an application that contains multiple Web applications, each of which contains its own copy of a session EJB named EJB1. If you want one MBean to collect information for all instances of the session EJB across all applications, you must package the MBean's classes in the `APP-INF` directory. If you want each Web application's copy of the EJB to maintain its own copy of the MBean, then package the MBean's classes in the EJB's JAR file. (If you package the classes in the EJB's JAR, then you distribute the MBean classes to each Web application when you copy the JAR to the Web application.)

CHAPTER 4

Instrumenting and Registering Custom MBeans

This section describes BEA's recommendation for instrumenting and registering standard MBeans for application modules. Figure 4-1 illustrates the process. The steps in the process, and the results of each are described in Table 4-1. Subsequent sections detail each step in the process.

Figure 4-1 Standard MBean Development Overview

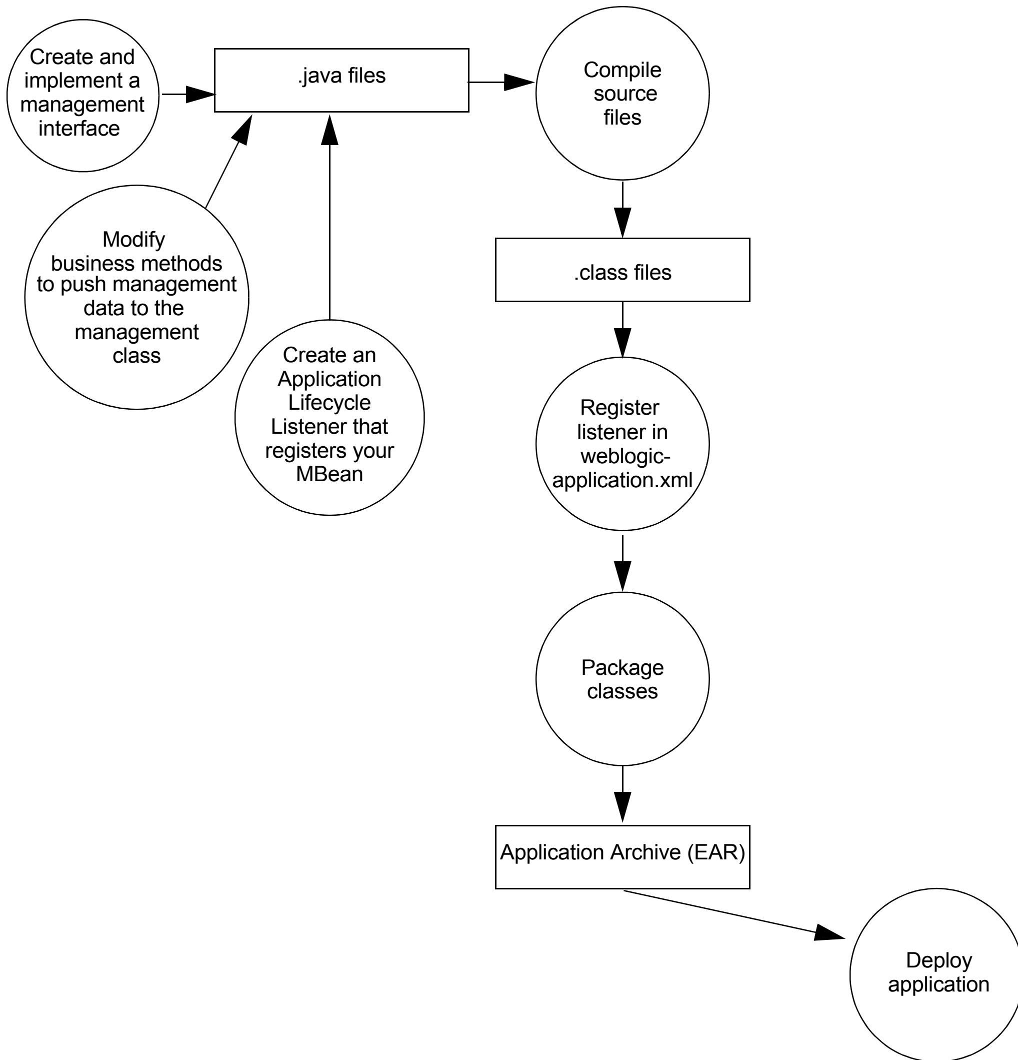


Table 4-1 Model MBean Development Tasks and Results

Step	Description	Result
1. "Create and Implement a Management Interface" on page 4-3	Create a standard Java interface that describes the properties (management attributes) and operations you want to expose to JMX clients. Create a Java class that implements the interface. Because management logic should be separate from business logic, the implementation should not be in the same class that contains your business methods.	Source files that describe and implement your management interface.
2. "Modify Business Methods to Push Data" on page 4-6	If your management attributes contain data about the number of times a business method has been invoked, or if you want management attributes to contain the same value as a business property, modify your business methods to push (update) data into the management implementation class. For example, if you want to keep track of how frequently your business class writes to the database, modify the business method that is responsible for writing to the database to also increment a counter property in your management implementation class. This design pattern enables you to insert a minimal amount of management code in your business code.	A clean separation between business logic and management logic.
3."Register the MBean" on page 4-7	If you want to instantiate your MBeans as part of application deployment, create a WebLogic Server `ApplicationLifecycleListener` class to register your MBean.	A Java class and added entries in `weblogic-application.xml`.
4. "Package Application and MBean Classes" on page 4-9	Package your compiled classes into a single archive.	A JAR, WAR, EAR file or other deployable archive file.

Create and Implement a Management Interface

One of the main advantages to the standard MBeans design pattern is that you define and implement management properties (attributes) as you would any Java property (using getter/is

and setter methods); similarly, you define and implement management methods (operations) as you would any Java method.

When you register the MBean, the MBean server examines the MBean interface and determines how to represent the data to JMX clients. Then, JMX clients use the `MBeanServerConnection.getAttribute()` and `setAttribute()` methods to get and set the values of attributes in your MBean and they use `MBeanServerConnection.invoke()` to invoke its operations. See `MBeanServerConnection` in the *J2SE 5.0 API Specification*.

To create an interface for your standard MBean:

1. Declare the interface as public.

2. BEA recommends that you name the interface as follows:
   `*Business-object*MBean.java`

   where `*Business-object*` is the object that is being managed.

   BEA's recommended design pattern for standard MBeans enables you to follow whatever naming convention you prefer. In other standard MBean design patterns (patterns in which the MBean's implementation file does not extend `javax.management.StandardMBean`), the file name must follow this pattern: `*Impl-file*MBean.java` where `*Impl-file*` is the name of the MBean's implementation file.

3. For each read-write attribute that you want to make available in your MBean, define a getter and setter method that follows this naming pattern:

   ```
   getAttribute-name
   setAttribute-name
   ```

   where `*Attribute-name*` is a case-sensitive name that you want to expose to JMX clients.

   If your coding conventions prefer that you use an `is*Attribute-name*` as the getter method for attributes of type `Boolean`, you may do so. However, JMX clients use the `MBeanServerConnection.getAttribute()` method to retrieve an attribute's value regardless of the attribute's data type; there is no `MBeanServerConnection.isAttribute()` method.

4. For each read-only attribute that you want to make available, define only an `is` or a getter method.

   For each write-only attribute, define only a setter method.

5. Define each management operation that you want to expose to JMX clients.

Listing 4-1 is an MBean interface that defines a read-only attribute of type `int` and an operation that JMX clients can use to set the value of the attribute to `0`.

Listing 4-1 Management Interface

```
package com.bea.medrec.controller;

public interface RecordSessionEJBMBean {
   public int getTotalRx();
   public void resetTotalRx();
}
```

To implement the interface:

1. Create a public class.

   BEA recommends the following pattern as a naming convention for implementation files: *`MBean-Interface`*`Impl.java`.

2. Extend `javax.management.StandardMBean` to enable this flexibility in the naming requirements.

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

3. Implement the `StandardMBean(Object implementation, Class mbeanInterface)` constructor.

   With BEA's recommended design pattern in which you separate the management logic into a delegate class, you must provide a public constructor that implements the `StandardMBean(Object implementation, Class mbeanInterface)` constructor.

4. Implement the methods that you defined in the management interface.

   Note the following guidelines:

   – If you are using BEA's recommend design pattern in which business objects push management data into the management object, provide a method in this implementation class that the business methods use to set the value of the management attribute. In Listing 4-2, the `incrementTotalRx()` method is available to business methods but it is not part of the management interface.

– If multiple instances of an EJB, servlet, or other class can set the value of a management attribute, make sure to increment the property atomically and do not make its getter and setter (or increment method) synchronized. While synchronizing guarantees the accuracy of management data, it blocks business threads until the management operation has completed.

Listing 4-2 MBean Implementation

```
package com.bea.medrec.controller;

import javax.management.StandardMBean;
import com.bea.medrec.controller.RecordSessionEJBMBean;

public class RecordSessionEJBMBeanImpl extends StandardMBean
   implements RecordSessionEJBMBean {

   public RecordSessionEJBMBeanImpl() throws
      javax.management.NotCompliantMBeanException {
         super(RecordSessionEJBMBean.class);
   }

   public int TotalRx = 0;
   public int getTotalRx() {
        return TotalRx;
   }
   public void incrementTotalRx() {
      TotalRx++;
   }
   public void resetTotalRx() {
       TotalRx = 0;
   }
}
```

Modify Business Methods to Push Data

If your management attributes contain data about the number of times a business method has been invoked, or if you want management attributes to contain the same value as a business property, modify your business methods to push (update) data into the management implementation class.

Listing 4-3 shows a method in an EJB that increments the integer in the `TotalRx` property each time the method is invoked.

Listing 4-3 EJB Method That Increments the Management Attribute

```
private Collection addRxs(Collection rXs, RecordLocal recordLocal)
   throws CreateException, Exception {
   ...
   com.bea.medrec.controller.RecordSessionEJBMBeanImpl.incrementTotalRx();
   ...
}
```

Register the MBean

If you want to instantiate your MBeans as part of application deployment, create an `ApplicationLifecycleListener` that registers your MBean when the application deploys (see "Use ApplicationLifecycleListener to Register Application MBeans" on page 3-4):

1. Create a class that extends `weblogic.application.ApplicationLifecycleListener`.
2. In this `ApplicationLifecycleListener` class, implement the `ApplicationLifecycleListener.postStart(ApplicationLifecycleEvent evt)` method.

   In your implementation of this method:

   a. Construct an object name for your MBean.

      BEA recommends this naming convention:
      `your.company``:Name=`*`Parent-module`*`,Type=`*`MBean-interface-classname`*

      To get the name of the parent module, use `ApplicationLifecycleEvent` to get an `ApplicationContext` object. Then use `ApplicationContext` to get the module's identification.

   b. Access the WebLogic Server Runtime MBean Server through JNDI.

      If the classes for the JMX client are part of a J2EE module, such as an EJB or Web application, then the JNDI name for the Runtime MBeanServer is:
      `java:comp/env/jmx/runtime`

If the classes for the JMX client are not part of a J2EE module, then the JNDI name for the Runtime MBean Server is:

`java:comp/jmx/runtime`

For example:

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

See Make Local Connections to the Runtime MBean Server in *Developing Custom Management Utilities with JMX*.

c. Register your MBean using `MBeanServer.registerMBean(Object object, ObjectName name)` where:

   `object` is an instance of your MBean implementation class.

   `name` is the JMX object name for your MBean.

When your application deploys, the WebLogic deployment service emits `ApplicationLifecycleEvent` notifications to all of its registered listeners. When the listener receives a `postStart` notification, it invokes its `postStart` method. See Programming Application Lifecycle Events in *Developing Applications with WebLogic Server*.

3. In the same class, implement the `ApplicationLifecycleListener.preStop(ApplicationLifecycleEvent evt)` method.

   In your implementation of this method, invoke the `javax.management.MBeanServer.unregister(ObjectName MBean-name)` method to unregister your MBean.

4. Register your class as an `ApplicationLifecycleListener` by adding the following element to your application's `weblogic-application.xml` file:

```
<listener>
   <listener-class>
      fully-qualified-class-name
   </listener-class>
</listener>
```

For an example of this technique, see the Medrec example server.

Package Application and MBean Classes

Package your MBean classes in the application's `APP-INF` directory or in a module's JAR, WAR or other type of archive file depending on the access that you want to enable for the MBean. See "Additional Design Considerations" on page 3-7.

Using the WebLogic Server JMX Timer Service

If you need your JMX client to carry out a task at a specified time or a regular time interval, you can configure a JMX timer service. The service emits notifications at specific dates and times or at a constant interval. Then you create a listener that listens for the timer notifications and responds appropriately.

For example, you want a JMX monitor to run between 9am and 9pm each day. You can use the JMX timer service to emit a notification each day at 9am, which triggers a JMX listener to start your monitor. Then the timer service emits another notification at 9pm, which triggers your listener to stop the monitor MBean.

The JDK includes an implementation of the JMX timer service (see `javax.management.timer.Timer` in the *J2SE 5.0 API Specification*), however listeners for this timer service run in their own thread in a server's JVM.

WebLogic Server includes an extension of the standard timer service that causes timer listeners to run in a thread that Weblogic Server manages and within the security context of a WebLogic Server user account.

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

- "Using the WebLogic Timer Service: Main Steps" on page 5-2
- "Example: Generating a Notification Every Five Minutes After 9 AM" on page 5-5
- "Removing Notifications" on page 5-8

Using the WebLogic Timer Service: Main Steps

Each JMX client that wants to use the timer service must construct and manage instances of the timer service as it requires (WebLogic Server does not provide a centralized timer service that all JMX clients use). Each time you restart a server instance, each JMX client must re-instantiate any timer service configurations it needs.

To use the WebLogic Server timer service:

1. Create a JMX listener class in your application.

   For general instructions on creating a JMX listener, see Creating a Notification Listener in *Developing Custom Management Utilities in JMX*.

2. Create a class that does the following:

   a. Configures an instance of `weblogic.management.timer.TimerMBean` to emit `javax.management.timer.TimerNotification` notifications at a specific time or at a recurring interval. See `TimerNotification` in the *J2SE 5.0 API Specification*.

      For each notification that you configure, include a `String` in the notification's `Type` attribute that identifies the event that caused the timer to emit the notification.

      See "Configuring a Timer MBean to Emit Notifications" on page 5-2.

   b. Registers your listener and an optional filter with the timer MBean that you configured.

   c. Starts the timer in the timer MBean that you configured.

      For general instructions, see Configuring a Notification Filter and Registering a Notification Listener and Filter in *Developing Custom Management Utilities in JMX*.

   d. Unregisters the timer MBean and closes it connection to the MBean server when it has finished using the timer service.

3. Package and deploy the listener and other JMX classes. See Packaging and Deploying Listeners on WebLogic Server in *Developing Custom Management Utilities in JMX*.

Configuring a Timer MBean to Emit Notifications

To configure a `Timer` MBean instance to emit a notification:

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

   See Connect to an MBean Server in *Developing Custom Management Utilities in JMX*.

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

   For example, `"mycompany:Name=myDailyTimer,Type=weblogicTimer"`

3. Create and register the timer MBean.

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

   – *`classname`* is `weblogic.management.timer.Timer`

   – *`name`* is the object name that you created for the timer MBean instance.

   Note: The time MBean that you create runs in the Java agent on WebLogic Server (it does not run in a client JVM even if you create the timer MBean from a remote JMX client).

4. Configure the timer MBean to emit a notification.

   Invoke the MBean's `addNotification` operation. Table 5-1 describes each parameter of the `addNotification` operation. For more information, see `weblogic.management.timer.Timer` in the *WebLogic Server API Reference*.

   The `addNotification` operation creates a `TimerNotification` object and returns an identifier (`Integer`) for the new `TimerNotification` object. You can use this identifier to retrieve information about the `TimerNotification` object from the timer or to remove the object from the timer's list of notifications.

5. Repeat step 4 for each timer notification that your JMX client needs to receive.

6. Start the timers in your timer MBean by invoking the timer MBean's `start()` operation.

When the time that you specify arrives, the timer service emits the `TimerNotification` object along with a reference to the handback object.

Table 5-1 Parameters of the addNotification Operation

Parameter	Description
`java.lang.String type`	A string that you use to identify the event that triggers this notification to be broadcast. For example, you can specify `midnight` for a notification that you configure to be broadcast each day at midnight.
`java.lang.String message`	Specifies the value of the `TimerNotification` object's `message` attribute.
`java.lang.Object userData`	Specifies the name of an object that contains whatever data you want to send to your listeners. Usually, you specify a reference to the class that registered the notification, which functions as a callback.
`java.util.Date startTime`	Specifies a `Date` object that contains the time and day at which the timer emits your notification. See "Creating Date Objects" on page 5-4.
`long period`	(Optional) Specifies the interval in milliseconds between notification occurrences. Repeating notifications are not enabled if this parameter is zero or is not defined (`null`).
`long nbOccurences`	(Optional) Specifies the total number of times that the notification will occur. If the value of this parameter is zero or is not defined (`null`) and if the period is not zero or `null`, then the notification will repeat indefinitely. If you specify this parameter, each time the `Timer` MBean emits the associated notification, it decrements the number of occurrences by one. You can use the timer MBean's `getNbOccurrences` operation to determine the number of occurrences that remain. When the number of occurrences reaches zero, the timer MBean removes the notification from its list of configured notifications.

Creating Date Objects

The constructor for the `java.util.Date` object initializes the object to represent the time at which you created the `Date` object measured to the nearest millisecond. To specify a different time or date:

1. Create an instance of `java.util.Calendar`.
2. Configure the fields in the `Calendar` object to represent the time or date.
3. Invoke the `Calendar` object's `getTime()` method, which returns a `Date` object that represents the time in the `Calendar` object.

For example, the following code configures a `Date` object that represents midnight:

```
java.util.Calendar cal = java.util.Calendar.getInstance();
cal.set(java.util.Calendar.HOUR_OF_DAY, 24);
java.util.Date morning = cal.getTime();
```

See `java.util.Calendar` in the *J2SE 5.0 API Specification*.

Example: Generating a Notification Every Five Minutes After 9 AM

The code in Listing 5-1 creates an instance of `weblogic.management.timer.Timer` that emits a notification every 5 minutes after 9am.

Note the following about the code:

- It creates and registers the timer MBean in the Runtime MBean Server, under the assumption that the JMX client runs alongside applications that are deployed on multiple server instances. In this case, your JMX client would register a timer MBean in each Runtime MBean Server in the domain.
- Even though it creates an instance of the WebLogic Server timer MBean, the class does not import WebLogic Server classes. Only the MBean server needs access to the WebLogic Server `Timer` class, not the JMX client.
- Any generic JMX listener can be used to listen for timer notifications, because all timer notifications extend `javax.management.Notification`.

Listing 5-1 Create, Register, and Configure a Timer MBean

```
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.NotificationFilterSupport;

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

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

   /*
    * Initialize connection to the Runtime MBean Server.
    * This MBean is the root of the runtime MBean hierarchy, and
    * 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.runtime";

      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];

   try {
      /* Invokes a custom method that establishes a connection to the
       * 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 timer MBean.
      ObjectName timerON = new
         ObjectName("mycompany:Name=myDailyTimer,Type=weblogicTimer");
      String classname = "weblogic.management.timer.Timer";
      connection.createMBean(classname, timerON);
      System.out.println("===> created timer mbean "+timerON);

      // Configures the timer MBean to emit a morning notification.
      // Assigns the return value of addNotification to a variable so that
      // it will be possible to invoke other operations for this specific
      // notification.
      java.util.Calendar cal = java.util.Calendar.getInstance();
      cal.set(java.util.Calendar.HOUR_OF_DAY, 9);
      java.util.Date morning = cal.getTime();
      String myData = "Timer notification";
      Integer morningTimerID = (Integer) connection.invoke(timerON,
         "addNotification",
         new Object[] { "mycompany.timer.notification.after9am" ,
         "After 9am!", myData, morning, new Long(60000) },
         new String[] {"java.lang.String", "java.lang.String",
         "java.lang.Object", "java.util.Date", "long" });

      //Instantiates your listener class and configures a filter to
      // forward only timer messages.
      MyListener listener = new MyListener();
      NotificationFilterSupport filter = new NotificationFilterSupport();
      filter.enableType("mycompany.timer");

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

```
         //Starts the timer.
         connection.invoke(timerON, "start", new Object[] { }, new String[] {});

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

Removing Notifications

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

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

The timer MBean also provides the following operations to remove notifications:

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

For more information, see `weblogic.management.timer.Timer` in the *WebLogic Server API Reference*.

Accessing Custom MBeans

Besides programmatic JMX access to your custom MBeans, you can use any JMX-compliant management system to access your MBeans. For information, see "What Management Services Have BEA Partners Developed?" on page 2-3 and the Sun Developer Network Web site, which provides links to books, white papers, and other information on JMX: http://java.sun.com/products/JavaManagement/.

The following sections describe additional ways to access your custom MBeans:

- "Accessing Custom MBeans from JConsole" on page 6-1
- "Accessing Custom MBeans from WebLogic Scripting Tool" on page 6-2
- "Accessing Custom MBeans from an Administration Console Extension" on page 6-2

Accessing Custom MBeans from JConsole

The JDK includes JConsole, a Swing-based JMX client that you can use to browse MBeans. You can browse the MBeans in any WebLogic Server MBean server and in the platform MBean server. Sun recommends that you use JConsole only in a development environment; it consumes significant amounts of resources. See Using JConsole to Monitor Applications at http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html.

To access custom MBeans from JConsole:

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. From a command prompt, make sure that JDK 1.5 or its equivalent is on the path.
3. In the command prompt, enter the following command: `jconsole`
4. If your custom MBeans are registered in the platform MBean server (or if you have configured the WebLogic Server Runtime MBean Server to be the platform MBean server):
   a. In the JConsole window, select Connection > New Connection.
   b. In the Connect to Agent window, select the Local tab and click Connect.
5. If your custom MBeans are registered in the WebLogic Server Runtime MBean Server, and if you have not configured the Runtime MBean Server to be the platform MBean server:
   a. In the JConsole window, select Connection > New Connection.
   b. In the Connect to Agent window, select the Advanced tab.
   c. On the Advanced tab, in the JMX URL box, enter:

      `service:jmx:rmi:///jndi/iiop://`*`host:port`*`/weblogic.management.mbeanservers.runtime`

      where *`host:port`* is the host name and port of the WebLogic Server instance that hosts your MBeans.

      For example:
      `service:jmx:rmi:///jndi/iiop://localhost:7001/weblogic.management.mbeanservers.runtime`
   d. In the User Name and Password boxes, enter the default IIOP user name and password.
   e. Click Connect.

Accessing Custom MBeans from WebLogic Scripting Tool

If you register your MBeans in the Runtime MBean Sever, you can use WebLogic Scripting Tool to access your custom MBeans. See Accessing Custom MBeans in *WebLogic Scripting Tool.*

Accessing Custom MBeans from an Administration Console Extension

You can extend the WebLogic Server Administration Console by creating Java Server Pages (JSPs) that conform to a specific template. Your JSP can include JMX code that connects to the

platform MBean server or the WebLogic Server Runtime MBean Server and looks up your MBeans.

For more information, see *Extending the Administration Console*.

Index

D

derived gauge, defined 2-7

J

JMX specification 1-2

M

monitor MBeans
    defined 2-7