BEA WebLogic Server

Introduction to WebLogic Server 5.1

Version 2.0
April, 2000

Introduction to BEA WebLogic Server

Document Edition	Part Number	Date	Software Version
2.0	860-001002-002	April 2000	WebLogic Server 5.1

Table of Contents

Preface

Purpose of This Document

This document describes BEA® WebLogic® Server, and provides an introduction to developing distributed applications with WebLogic Server 5.1.

Who Should Read This Document

This document is for anyone who wants to build e-commerce applications using WebLogic Server, including developers, system administrators, and network administrators. It assumes a familiarity with Java programming and web-based applications.

How This Document is Organized

Introduction to BEA WebLogic Server is organized as follows:

- Chapter 1, “Introduction,” introduces WebLogic Server concepts, including WebLogic Server architecture, application servers, and multitier application architecture.
- Chapter 2, “WebLogic Server Administration,” introduces WebLogic Server security facilities, administrative tools and utilities, and WebLogic clusters.
- Chapter 3, “WebLogic Server Developer APIs,” introduces the WebLogic Server development environment and the examples that appear in this document. The remaining chapters are tutorials for the primary WebLogic Server application development facilities. Each chapter describes a technology and presents an example that you can compile and run:
  - Chapter 4, “Servlets”
  - Chapter 5, “JSP (JavaServer Pages)”
  - Chapter 6, “JDBC (Java Database Connectivity)”
  - Chapter 7, “EJB (Enterprise JavaBeans)”
  - Chapter 8, “JMS (Java Message Service)”

How to Use This Document

This document, *Introduction to BEA WebLogic Server*, is available in hardcopy and online in PDF format. You can print this document using Adobe Acrobat. Retrieve the PDF file from the Internet at *http://www.weblogic.com/docs51/API_users_guide.html*. If you have WebLogic Server 5.1 on CD-ROM, the file is on the CD-ROM at */documents/docs/intro/intro.pdf*.

Related Documents

The documentation for WebLogic Server can be found on the Internet at the URL *http://www.weblogic.com/docs51/index.html*. If you have WebLogic Server 5.1 on CD-ROM, you can view the documentation with a browser by loading the */documentation/docs/index.html* file from your CD-ROM.

The documentation on our Web site is updated frequently. You can always find the most current documentation by browsing our Web site.

Contact Information

The following sections provide information about how to obtain support for the documentation and software.

Documentation Support

If you have questions or comments on the documentation, you can contact us by e-mail at *docsupport@bea.com*.

Customer Support

If you have any questions about this version of WebLogic Server, or if you have problems installing and running WebLogic Server, contact Customer Support through BEA WebSupport at *http://www.bea.com*. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Introduction

Introduction to BEA WebLogic Server introduces you to *WebLogic Server,* BEA's Web Application Server. This chapter, "Introduction," covers the following topics:

- What is WebLogic Server?
- Multitier Application Architecture
- WebLogic Server Architecture
- WebLogic Server and Sun's Java 2 Platform
- WebLogic Server Application Models
- Java Development with WebLogic Server

What is WebLogic Server?

Today's business environment demands web and e-commerce applications that will accelerate your entry into new markets, help you find new ways to reach and retain customers, and allow you to introduce new products and services quickly. To build and deploy these new solutions, you need a proven, reliable e-commerce platform. One that can connect and empower all types of users, while integrating all of your corporate data, mainframe applications, and other enterprise applications in a powerful, flexible end-to-end e-commerce solution. One with the performance, scalability, and high availability to handle your most critical enterprise-scale computing.

As the industry leading e-commerce transaction platform, WebLogic Server provides a number of features critical to developing and deploying mission-critical e-commerce applications across distributed, heterogeneous computing environments. These include:

- Standards leadership—Comprehensive Enterprise Java support, including EJB and JMS, to ease the implementation and deployment of application components.
- Enterprise e-business scalability—WebLogic Server's software clustering of dynamic web pages (servlets, JavaServer Pages) and EJB business components, coupled with client connection sharing and database resource pooling, maximize efficiency in the use of critical resources.
- Rich client options—WebLogic Server supports direct HTTP requests from browsers, HTTP servlets to drive dynamic HTML content, and JavaServer Pages (JSP) for managing the online update of dynamic web pages.
- Robust administration—WebLogic Server offers a comprehensive pure-Java console, Zero Administration Client (ZAC) for managing the distribution of applications to remote users, and dynamic application partitioning and cluster membership.
- E-commerce-ready security—WebLogic Server features Secure Sockets Layer (SSL) support for integration of encryption and authentication security into e-commerce solutions.
- Maximum development and deployment flexibility—WebLogic Server features tight integration with and support for leading databases, development tools, and other environments.

WebLogic Server allows you to quickly develop and deploy reliable, secure, scaleable and manageable applications. WebLogic Server manages system-level details so you can concentrate on business logic and presentation.

WebLogic Server operates at the center of a multitier architecture. In this architecture, business logic is executed in WebLogic Server, rather than in client applications. The resulting “thin” client, three tier architecture allows the client to manage the presentation layer, the application server to manage the business logic and the back end data services manage the data. This makes WebLogic Server the ideal platform for web-enabled e-commerce applications.

In the middle tier, WebLogic Server provides a reliable, highly scalable platform for hosting business logic. It serves static and dynamic web pages, and manages database access, security, and transaction services for applications. WebLogic Server centralizes access to a variety of third-tier resources and back end services. Tier three services include databases, messaging systems, transaction monitors, real-time data feeds, and existing enterprise information systems integrated with WebLogic Server via “connectors.” WebLogic Server shields client applications from propriety interfaces and provides efficient sharing of resources.

Managing access to critical back-end resources from the middle tier helps to secure them. In a web-based application, all client interaction is accomplished with HTML web pages. This means that clients outside of a firewall can access a WebLogic Server application without compromising the security of the back-end resources the application uses. Java-based WebLogic Server applications provide similar security for back-end resources through the use of components located in the middle tier.

Instead of focussing on all of the complexities of system infrastructure, WebLogic Server developers focus on modeling business processes and solving application needs.

Multitier Application Architecture

With the traditional two-tier client-server architecture, a server provides a specific service and provides the client with an interface to access that service. The server performs low-level, computation- and resource-intensive tasks on behalf of its clients. The server also provides optimization, resource management, and concurrency control for its multiple clients. This is a considerable improvement over an independent client architecture where each client must implement complex services and somehow share resources cooperatively with other clients.

With the client-server architecture, however, clients use server-specific, client-side APIs to access a server, and each API provides access to a single category of services, such as database or messaging. Client-side APIs are proprietary and vendor-specific. The client code and vendor libraries must be installed and maintained on every computer where the client application is to be used.

In spite of its advantages, the client-server architecture still produces client-side applications that are large, complex, platform- and vendor-specific, and difficult to deploy and maintain.

A multitier architecture based on open standards preserves the advantages of the client-server architecture while simplifying the complexity by adding an additional tier between clients and the services they access. The middle tier can manage client-server resources more efficiently and provide greater scalability, using techniques such as resource-pooling and caching. It presents clients with an abstracted API that integrates back-end resources and shields clients from the back-end vendors’ implementation specifics. Client applications are easier to deploy because they are unaffected by changes in the back-end.

An application server can further improve on the client-server architecture by moving much or all of the application logic from the client into the middle tier, so that clients concentrate exclusively on presentation logic.

With a WebLogic Server application deployed to the web, for example, all of the application-specific logic can be located in WebLogic Server; clients invoke applications by requesting web pages and providing input through HTML forms. Servlets or JavaServer Pages, running on WebLogic Server, accept input from the browser and provide the content for the browser.

Servlets and JSPs can access a variety of other services in WebLogic Server. They can execute database queries using standard JDBC (Java Database Connectivity) calls or Enterprise JavaBeans. They can invoke Enterprise JavaBeans that encapsulate specific business logic, and they can use JMS (Java Messaging Service) to exchange messages with other clients and applications.

With a Java client application, the application logic can be selectively deployed in the client or in WebLogic Server. Java clients are typically deployed to overcome the limitations of browsers access to resources on the client computer and less dynamic user interfaces.

WebLogic Server Architecture

WebLogic Server is a multithreaded server that listens on the network for a client request. It establishes a connection with a client, including negotiating details such as protocol, encryption, and authentication. Then it processes the client's requests by executing Java classes on behalf of the client.

On the back-end, WebLogic Server can connect to virtually any network-accessible service. For example, nearly all applications require database access. Other common back-end services are directory and naming services, messaging systems, and legacy applications that are integrated with WebLogic Server applications.

WebLogic Server can be a primary web server or it can process requests redirected to it by an existing web server. For example, it is common to have a Netscape Enterprise Server, Microsoft Internet Information Server, or Apache web server handle requests for static HTML web pages, but pass their servlet and JSP page requests to WebLogic Server.

WebLogic Server clients can be web browsers using HTTP (Hypertext Transfer Protocol), or Java clients. In both cases, clients specify a URL (Universal Resource

Locator), which provides the network protocol and location of the desired WebLogic Server resource. Secure, encrypted connections can be made using SSL (Secure Sockets Layer).

WebLogic Server provides scalability and reliability with WebLogic clusters. A WebLogic cluster is a group of cooperating WebLogic Servers that provide load-balancing and failover capabilities. Client requests are distributed among the servers participating in the cluster. When a WebLogic Server does not respond to a request, the client tries the next WebLogic Server. WebLogic EJB components and RMI classes support pluggable load-balancing algorithms. A WebLogic Server cluster can also work cooperatively with intelligent networking software, hardware, or proxy servers to provide even more load-balancing options.

WebLogic Server and Sun's Java 2 Platform

WebLogic Server implements Sun Microsystems' Java 2 Platform, Enterprise Edition (J2EE), a standard architecture for distributed, mutitier enterprise applications. WebLogic Server is the most complete J2EE application server available.

Sun's J2EE architecture emphasizes thin clients with application logic implemented in software components hosted in the middle tier. J2EE defines a suite of standard services that support components so that application logic does not have to be concerned with low-level details such as networking, database access, security, and data-sharing. WebLogic Server provides these services in a standard way so that application programmers can concentrate on business logic and presentation.

Component technologies include servlets and JavaServer Pages (JSP) for server-side presentation logic, and Enterprise JavaBeans (EJB) for implementing business objects and processes.

J2EE services that support components include:

- Java Naming and Directory Interface (JNDI), for accessing components and services
- Java Database Connectivity (JDBC), for moving database data to and from components
- Remote Method Invocation (RMI), for accessing components outside of the middle tier
- Access Control Lists (ACLs), for controlling access to components and applications

- Java Transaction Service (JTS) and Java Transaction API (JTA), for managing transactions
- Java Message Service (JMS), for implementing message-based applications

To promote fast development and portability, J2EE identifies common services needed by components and implements them in the container that hosts the component. A component has only the code necessary to describe the object or process that it models. It has no code to access its execution environment or services such as transaction management, access control, network communications, or persistence mechanisms. These services are provided by the "container," which is implemented in WebLogic Server.

This component/container abstraction allows developers to work within their fields of expertise. EJB developers create Enterprise beans that model business objects and processes within their own field of knowledge—for example accounting systems, human resources, or order processing. Web engineers design dynamic web pages that access beans by using simple JSP extensions to HTML.

J2EE also removes non-developer roles from component design by defining the roles of application assemblers and deployers. For example, mapping an enterprise bean to a particular database or other persistent store is a deployment responsibility, not a responsibility of the EJB developer. These services are configured at deployment and implemented by the container—WebLogic Server.

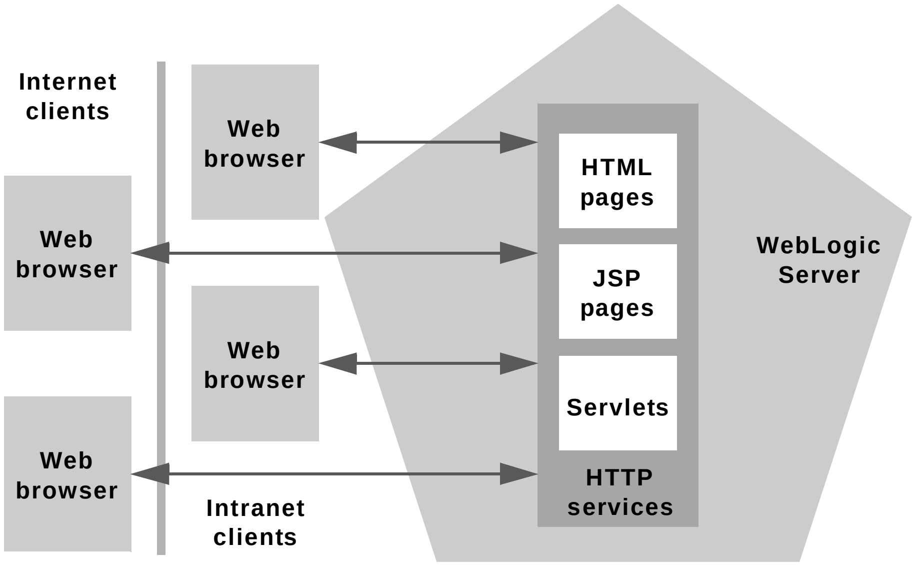
WebLogic Server Application Models

This section describes some common WebLogic Server application models. To use the the multitier architecture to its fullest advantage, the models emphasize placing business logic on WebLogic Server and presentation logic on thin clients, and using server-side components to minimize coding and take advantage of WebLogic Server's powerful component management services.

Web Services Application Model

A web-based application takes advantage of WebLogic Server's web server capabilities. Using HTTP, web browsers and other HTTP servers can make requests to WebLogic Server. WebLogic Server can serve static HTML pages, XML documents, Java Applets, and dynamic HTML pages generated by servlets and JSP pages. All application logic is located in WebLogic Server, so clients are very easy to deploy. Figure 1-1 illustrates the architecture.

Figure 1-1 Web services application model



Web applications communicate using HTTP. With HTTP, clients transmit a request to the server and the server returns a response. The protocol defines several types of requests, GET and POST are the most common. A GET request asks the server for a page at a specified URL. A POST request sends the contents of an HTML form to the server for processing. In either type of request, the server returns an HTML page to the browser.

Static HTML Pages and Applets

Static HTML pages are text files that contain text formatted with standard HTML tags. When a client requests a static HTML page with an HTTP GET request, WebLogic Server returns the contents of the file to the client as an HTTP response. This is a standard service available from any web server.

An applet is a Java class that executes in a web browser. To WebLogic Server, an applet is a file that is included in the body of a static HTML page, much like an image. The web browser executes the applet in its own Java interpreter. Applets are often used to visually activate a web page or to enhance the user interface and presentation logic on the client.

Servlets

Servlets are Java programs that reside on WebLogic Server. When a client requests a servlet, WebLogic Server executes the servlet on the client's behalf. The servlet generates a response with an HTML or XML body. To the client, the response from a servlet looks like any static web page. However, the contents of the page returned can be different each time the servlet executes.

A servlet can access data sent with the client's request, such as values entered into HTML form fields. A servlet can also execute other Java classes in WebLogic Server and include the results in its response.

JSP Pages

JSP pages are files that contain text formatted with standard HTML tags, and JSP tags and scriptlets. JSP pages are based on servlet technology. JSP tags provide access to data included in the client's HTTP request and to other WebLogic Server classes, including JavaBeans and Enterprise JavaBeans. JSP scriptlets contain Java code that is included in the generated servlet.

When a client requests a JSP page, the WebLogic JSP compiler translates the embedded JSP tags and scriptlets and generates a servlet. A page is only compiled the first time it is requested and when it has been revised.

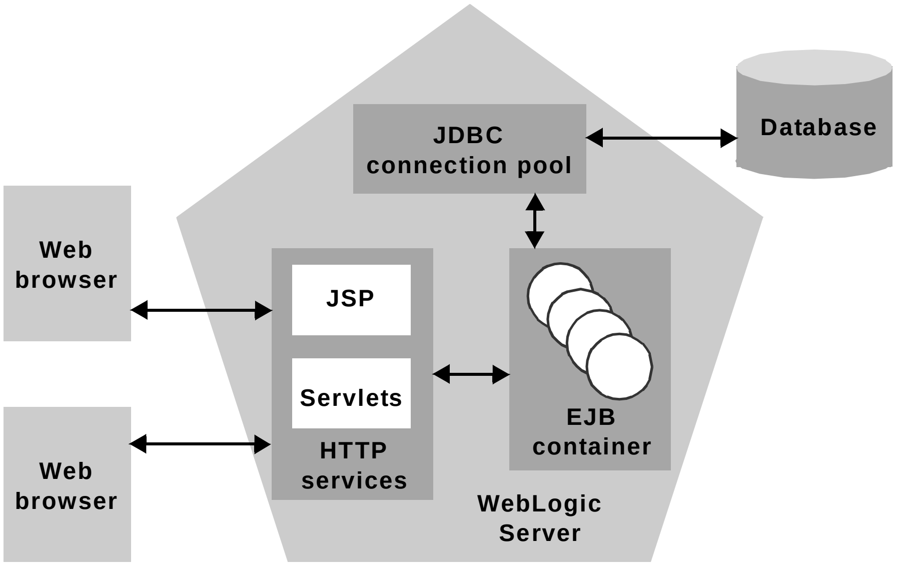
JSP pages are more convenient to create than servlets because you can concentrate on page design using familiar HTML tags. This is in keeping with the J2EE model of tailoring work to the developer's domain of expertise; HTML is a more suitable medium for a page designer than is Java. The Java code embedded in a JSP page can be minimized by referencing JavaBean and Enterprise JavaBean components on the page, rather than writing logic into JSP scriptlets.

Web-based Component Application Model

In a web-based component application, a web browser client accesses servlets and JSP pages on WebLogic Server. The servlet or JSP page, in turn, accesses JavaBeans, Enterprise JavaBeans, and other services on WebLogic Server. Since no code is deployed on the client, the client is thin. The client and server exchange requests and responses over HTTP.

Figure 1-2 illustrates the model.

Figure 1-2 Web-based component application model



The web-based component application model extends the web services application model by adding EJB components to the middle tier. In Figure 1-2, a JDBC connection pool is also present in the middle tier to provide pooled access to a database.

This application model maintains the security, manageability, and deployment advantages of a thin client. There are performance advantages to co-locating the server-side presentation logic (servlets and JSP pages) and EJB components on the same server. The network activity is confined to the HTTP connection between the client and WebLogic Server.

Enterprise JavaBeans

EJB (Enterprise JavaBeans) is the standard component architecture for distributed Java applications. The EJB architecture simplifies developing and using components by insulating beans from the system-level services that support them. EJB defines the responsibilities of a container, which is implemented by the WebLogic EJB container. The container takes care of such tasks as creating and destroying bean instances, managing transactions and concurrency, and loading and saving bean data to a persistent store. The runtime characteristics of a bean are configured when the bean is deployed in WebLogic Server, not when it is created by the EJB developer.

The EJB model allows bean developers to concentrate on pure business logic without concern for the underlying system support. This speeds the development process and makes it possible for application developers to use beans in different environments without having to modify the Java code in the bean.

There are two types of enterprise beans: session beans and entity beans. A session bean is associated with one client at any time and contains procedural logic. Session beans provide services such as performing calculations and database queries for a client or managing a set of related entity beans for a client. A session bean can be stateless, which means that it maintains no client-related data, or it can be stateful, which means that it maintains a conversational state with the client.

An entity bean represents a data object, such as a catalog item, customer, or invoice. An entity bean has a primary key, which ensures that a single bean instance represents an individual business object in WebLogic Server. Entity beans are mapped to data in a persistent data store, such as a relational database or object database. Mapping a bean to a data store is a task performed when a bean is deployed. In Figure 1-2, entity beans are persisted in a relational database via a WebLogic Server JDBC connection pool. WebLogic EJB includes built-in support for this type of persistence. To use another store, you can provide your own persistence code, or use tools provided with the data store. For example, TOPLink for BEA WebLogic Server, from *The Object People* (*http://www.objectpeople.com*), provides integrated support for EJB persistence in a high-performance object relational database.

JDBC Connection Pools

Nearly all applications require database access. In Java, database access is accomplished using JDBC (Java Database Connectivity). JDBC provides a standard interface for executing SQL commands and processing results in Java programs. A database vendor-specific JDBC driver loaded at runtime makes it possible to write Java applications that are portable between database vendors. BEA supplies JDBC drivers for Oracle, Microsoft SQL Server, and Informix databases. The Cloudscape JDBC driver and the Sybase jConnect driver are bundled with WebLogic Server. You can also use any other standard JDBC driver with WebLogic Server.

Creating a connection to a database is an expensive process. For short database transactions, more system resources may be consumed connecting to the database than executing the transaction. The multitier architecture offers a solution to this in the form of JDBC connection pools. With a connection pool, a set of database connections are established once when WebLogic Server starts up and then connections can be borrowed from the pool as needed.

In the web-based component model, Enterprise JavaBeans can perform all of the JDBC access for applications. Entity beans that use JDBC persistence use the JDBC pool to save their values. Database queries can be performed using session beans.

A servlet or JSP page can also borrow a connection from the connection pool to perform a database operation without the assistance of EJB. But using EJB allows you to create shareable database access components that are managed by the EJB container and easily referenced in JSP pages.

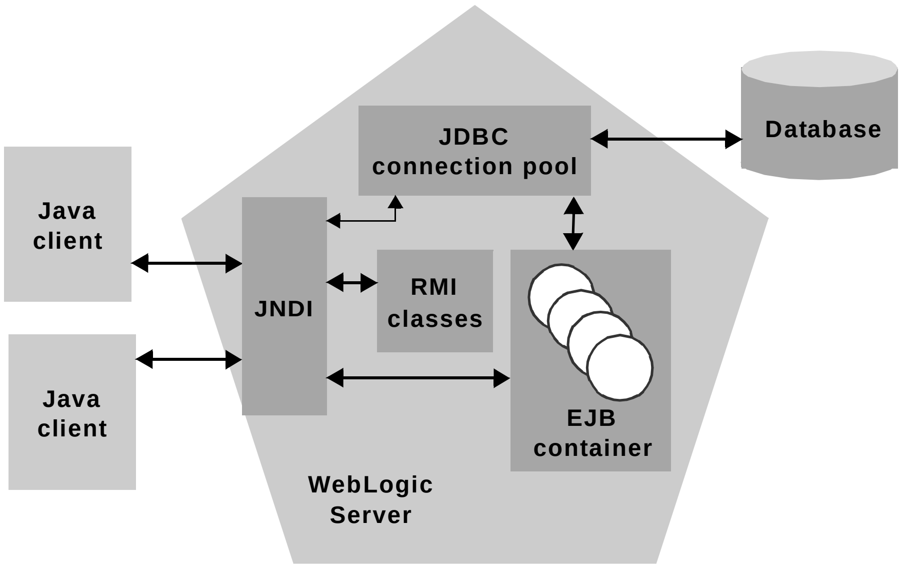
Java Client Application Model

Standalone Java clients can access WebLogic Server resources, including Enterprise JavaBeans and JDBC connection pools. They can execute Java classes on WebLogic Server using RMI (Remote Method Invocation). Presentation logic must be handled by the Java client, perhaps using JFC (Java Foundation Classes) to build interactive graphical interfaces.

Java clients, by definition, require client-side code. They also require a JRE (Java runtime environment) and WebLogic Server classes installed on the client computer. Because of these deployment requirements, Java clients are more appropriate for enterprise applications than e-commerce applications, which should run solely in the client's web browser.

Java clients are not constrained by the browser security model, which prohibits some applets from accessing resources on the client's computer. For example, a good application for a Java client is transferring data between a mobile database and an enterprise database via a WebLogic Server connection pool.

Figure 1-3 Java client application model



Java clients use JNDI (Java Naming and Directory Interface) to gain access to WebLogic Server. WebLogic Server maintains a JNDI naming tree where all remotely-executable classes, EJBs, and other server-side resources are registered. Using JNDI calls, the client looks up the desired service and receives a “stub” from WebLogic Server. A stub is a light-weight client-side representation of a server-side object. The client makes a request on the stub and the RMI framework transparently transfers the request to the server and returns the results to the client.

Java Development with WebLogic Server

WebLogic Server implements a number of Sun Java specifications, each describing a technology in detail. Many of the Sun specifications include both client-side and server-side APIs. WebLogic Server implements the server-side classes of the specifications and you, the application developer, implement your application using Sun APIs. You only use WebLogic APIs if you want to use one of our proprietary APIs—WebLogic Events, for example—or in a few cases where we have extended a Sun API to support features missing from the specification.

For example, with the Java Servlet API, you can create an HTTP Servlet that interacts with a web browser, producing dynamic web pages. The Java Servlet specification defines a “container,” whose responsibilities are to provide network services between the Servlet and the browser and to manage the Servlet through its lifecycle. In other words, a Servlet requires a Servlet engine to run it. WebLogic Server provides the Servlet engine.

Most Sun APIs have similar server-side/client-side implementation requirements. Sometimes, the server-side implementation is called a “container”, other times it is called a “service provider.”

Some Java APIs, for example JDBC (Java Database Connectivity), require no server-side implementation. A Java application can use JDBC to access a database without an application server in the middle tier. But the JDBC support implemented in WebLogic Server offers clear advantages, such as JDBC connection pools.

Also, because WebLogic Server implements all of the Enterprise Java APIs, applications hosted on WebLogic Server can access multiple services within a single JVM. For example, a servlet can use Enterprise JavaBeans, execute JDBC queries against a database, and send and receive JMS messages within a single WebLogic Server instance. This provides significant scalability and performance advantages. Web-based applications have access to a variety of Java client APIs without ever running Java code on the client.

Even with Java code running on the client, applications can benefit by concentrating their business logic in the middle tier. And since all requests from a Java client to WebLogic Server share a single connection to the server, network resources are used efficiently.

What’s Next?

Here are some online resources to help you learn more about the Java language:

- Take the *Java Tutorial* at *http://java.sun.com/docs/books/tutorial*
- Join the *Java Developers Network* at *http://developer.java.sun.com/developer*
- *jGuru*, at *http://www.jguru.com*, offers Java training and certification programs and a variety of other resources for Java developers

The next chapter describes WebLogic Server administration, including security features, administration tools and WebLogic Server clusters.

CHAPTER

2 WebLogic Server Administration

In addition to providing a rich runtime environment for multitier applications, an application server must be robust, scalable, and secure. This chapter describes WebLogic Server facilities for security, monitoring and tuning performance, and for clusters, which make it possible to deploy multiple WebLogic Servers to support increasing numbers of clients with failover capabilities.

This chapter covers the following topics:

- WebLogic Server Administrative Facilities
- The WebLogic Console
- WebLogic Server Security
- Performance Packs
- WebLogic Server Clusters

WebLogic Server Administrative Facilities

WebLogic Server includes a variety of administrative tools. The `weblogic.properties` file, a Java properties file, is the primary method for configuring WebLogic Server. The WebLogic Console is a graphical administration tool you can use to monitor and manage a running WebLogic Server. You can view WebLogic Server administrative information in a web browser using the Admin servlets. Finally, WebLogic Server includes several command line utilities that you can use to administer the server.

Configuring WebLogic Server

WebLogic Server is primarily configured with a Java properties file, named `weblogic.properties`. The file is located in the WebLogic Server installation directory. If you use WebLogic clusters, there is one cluster-wide properties file and an optional individual per-server properties file for each WebLogic Server participating in the cluster. See "WebLogic Server Clusters" in this chapter for more about clusters.

WebLogic Server is distributed with a default `weblogic.properties` file with properties set so that you can get WebLogic Server running quickly. You can edit the `weblogic.properties` file with any text editor that saves pure ASCII text.

If you installed with the zip file distribution, you *must* edit the file and set the password for the "system" user before you can run WebLogic Server. You should also search the file for file and directory names and edit them for your environment.

You can learn about many properties by exploring the properties file. It is heavily commented, and includes many properties that are described, but commented-out. You can find online documentation for all WebLogic Server properties in *Setting WebLogic Properties* at *http://www.weblogic.com/docs51/admindocs/properties.html*.

The WebLogic Console

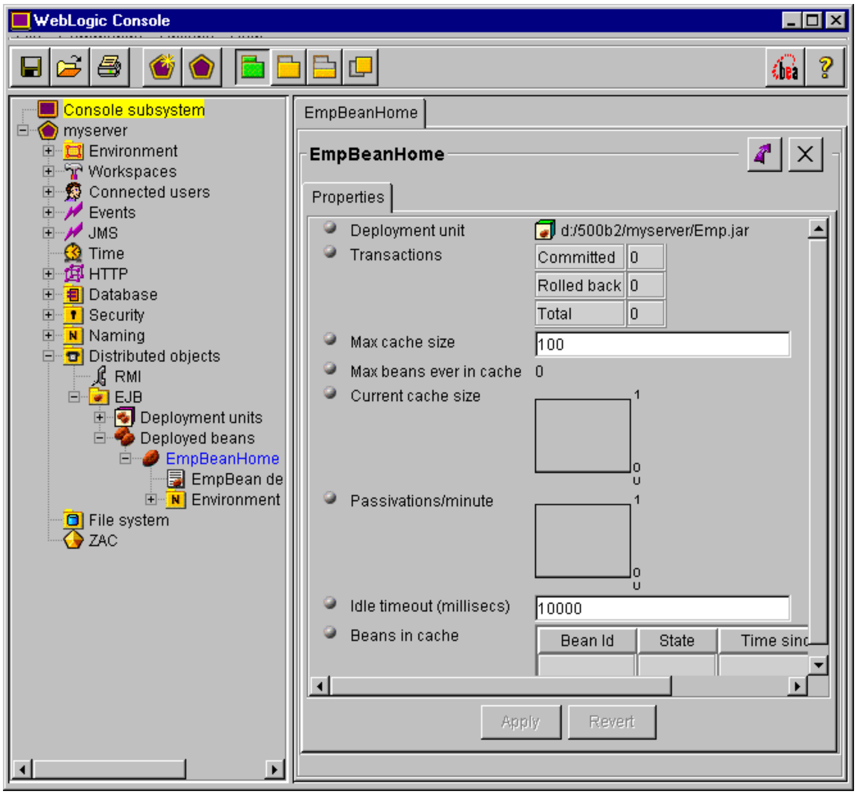
The WebLogic Console is a Java application that you can use to monitor and administer WebLogic Server. Here are some of the things you can do with the WebLogic Console:

- Start or stop WebLogic Server
- Temporarily disable and re-enable WebLogic Server logins
- Force garbage collection in the WebLogic Server JVM
- View WebLogic Server configuration properties, including properties set in the properties file, defined on the `weblogic.Server` command line, and set internally by WebLogic Server
- View WebLogic licenses installed in your server
- View the status of each WebLogic Server thread

- See messages in the WebLogic Server log, the HTTP log, and the JDBC log
- View information about connected users
- View status of, and statistics for, WebLogic Server resources, including workspaces, events, HTTP, JNDI, JMS, EJB, JDBC, ZAC, security, and file system objects
- Destroy, undeploy, or redeploy many objects such as connection pools, JMS Queues and Topics, EJBs and servlets
- Reset JDBC connection pools

The WebLogic Console displays a hierarchical tree in its left pane and information about the selected object in the tree in its right pane. The right pane has a Properties tab that displays information about the selected object. Some objects have a Commands tab that you can use to administer the object.

Here is an example of the Console's display for the Emp Enterprise JavaBean example:



EmpBeanHome

Deployment unit	d:/500b2/myserver/Emp.jar
Transactions	Committed 0
	Rolled back 0
	Total 0
Max cache size	100
Max beans ever in cache	0
Current cache size	
Passivations/minute	
Idle timeout (millisecs)	10000
Beans in cache	Bean Id / State / Time sinc

See *Using the WebLogic Console* at *http://www.weblogic.com/docs51/admindocs/console.html* for help using the Console.

WebLogic Server Security

WebLogic Server provides a comprehensive security architecture encompassing access control, cryptography-based privacy, and user authentication.

User-level and group level ACLs (access control lists), Realms, SSL (Secure Sockets Layer), and digital certificates are all standards-based security measures used in the

WebLogic Server. Used together, these security features track who has access to what services. A developer can restrict access to some WebLogic services through application logic at design time and the system administrator can also define how services are accessed at deployment. Additionally, WebLogic Server can operate independently from other security services or be incorporated into a single-sign-on solution by accessing existing security information stores.

Specifying security through the properties file allows easy prototyping and allows basic production systems to be up and running very quickly. For installations with more complex security needs, WebLogic Server includes support for integrating with other security services.

Users and Groups

WebLogic Server security begins with a specification of users. Users are entities that will connect to WebLogic Server and access WebLogic services. Users may be human end-users, other applications, or even other servers, such as web servers, proxy servers, or WebLogic Servers.

In the simplest case, the WebLogic Server administrator defines the names and passwords of users in the `weblogic.properties` file, which WebLogic Server reads at startup time. Users can be combined into groups for role-based authorization by setting additional properties in the file.

Defining Users

You define a user with a `weblogic.password` property. For example, to add the WebLogic user “joeuser” with password “joesPassword,” you add this property to the `weblogic.properties` file:

```
weblogic.password.joeuser=joesPassword
```

WebLogic Server passwords must have at least 8 characters. You can require longer passwords by changing the `weblogic.system.minPasswordLen` property in the `weblogic.properties` file.

Defining Groups

A group allows you to assign permissions to groups of users. You create a group by adding a `weblogic.security.group` property in the `weblogic.properties` file. For example, to create a “managers” group containing users “bill,” “ed,”, and “al” you first define the users, and then the group, using properties like these:

```
weblogic.password.bill=billsPassword
weblogic.password.ed=PasswordOfEd
weblogic.password.al=alsPassword
weblogic.security.group.managers=bill,ed,al
```

Special Users and Groups

WebLogic Server defines two special users, “system” and “guest.” The password for the “system” user must be defined in the `weblogic.properties` file before you can start WebLogic Server. If you install WebLogic Server on Windows NT using the InstallShield distribution, the “system” password is set as part of the installation process. Otherwise, you must edit the `weblogic.properties` file and set the password before you can run WebLogic Server.

The “system” user has administrative privileges in WebLogic Server and is needed to initialize internal services when WebLogic Server starts up. The “system” user account must also be used to perform activities such as shutting down WebLogic Server. The default `weblogic.properties` file restricts some capabilities to the “system” user.

The “guest” user is the default identity for unauthenticated WebLogic users. If a user does not provide a specific identity when connecting, WebLogic Server assigns the “guest” identity and permits only those activities allowed for the “guest” user. You do not have to define the “guest” user; it is created automatically (with the password “guest”). You can add a `weblogic.password.guest` property to change this user’s password.

WebLogic Server provides a default “everyone” group, which includes all users. Permissions assigned to this group are available to any user.

Authentication

Since security requirements vary greatly, WebLogic Server is configurable to allow different levels of authentication depending upon the needs of the deployment. When a client, administrator, or peer wants to access a protected WebLogic resource, client authentication is required. When a browser wants to establish a secure connection to WebLogic Server, server authentication is required.

WebLogic Server supports two mechanisms, defined by the HTTP protocol, to authenticate users: basic (username and password) and strong (digital certificates). Developers can create application-specific authentication mechanisms, which is a common feature of many web-based commercial applications.

Basic Authentication

Basic authentication is accomplished by requiring users to supply their usernames and passwords for authentication. When a web browser user requests a protected HTML page, HTTP Servlet, or JSP page, the browser displays a dialog box requesting a username and password. WebLogic Server passes the username and password to the WebLogic security realm for authentication.

A Java application performs basic authentication by including the username and password with the information it sends to WebLogic Server to establish a JNDI connection.

Some resources are intentionally unprotected—public HTML pages, for example. For unprotected resources, WebLogic Server allows unauthenticated users to assume the “guest” user identity within the WebLogic security realm. This is transparent to end users.

Strong Authentication

Strong authentication is accomplished by establishing a secure connection using SSL (Secure Sockets Layer). SSL is a standard developed by Netscape for secure connections on the web.

SSL defines a protocol that allows clients and servers to authenticate each other using certificates, and then to negotiate algorithms for encrypting the data they exchange. The certificates are based on public key encryption. With public key encryption, two files are generated for a user or server: a public key and a private key. Anyone who has the public key can encrypt data and transmit it. But the data can only be decrypted using the corresponding private key. Therefore, the private key is always stored locally—installed in a user’s browser or on their local hard disk, or in a server’s file system—and never transmitted over the network.

To obtain a certificate, a request containing the public key is submitted to a trusted third party called a Certificate Authority (CA). WebLogic Server supports the VeriSign and GTE CyberTrust CAs out-of-the-box. The CA verifies the identity of the user or server and returns a digitally-signed certificate that includes the public key and the CA’s certificate.

Most web-based applications use certificates to authenticate the server (web server or application server) to the client (web browser). By checking the server’s certificate, the client verifies that it has connected to the intended server. A web browser checks the server’s certificate to ensure that it was issued by a recognized CA, that it has not expired, and that the server is running at the DNS name specified in the certificate. The web browser alerts the user when a problem is detected with the certificate.

Applications that require authenticated clients generally use password authentication rather than requiring users to present certificates. During the SSL negotiation, the server's certificate is authenticated and a secure, encrypted connection is established between the server and client. Then the client's username and password can be obtained using the basic authentication mechanism, and transmitted to the server securely.

For very high-security enterprise applications, WebLogic Server can be configured to require clients to present certificates issued by selected CAs. This requires every user to obtain a certificate from one of the approved CAs.

Authenticating a client certificate does not provide an identity for a client; it just establishes the client as a trusted entity. Although the client's identity can be extracted from the certificate, this is not an automatic side effect of the SSL negotiation. The WebLogic Server distribution includes an example to demonstrate how to extract a user's identity from their certificate. When a WebLogic Server user identity is required and none has been supplied, WebLogic Server initiates basic authentication, even when the client has been authenticated with a certificate.

Authorization

WebLogic Server uses the Java ACL standard, distributed with JDK 1.1 and later, to extend the security framework of Java and make it practical for use at the enterprise level.

An ACL (Access Control List) is a structure with three parts:

1. a resource, or class of resources to protect
2. one or more permissions that can be granted on the resource or resources
3. a list of users and groups who are granted each permission

A permission is the ability to carry out a certain operation on a WebLogic Server resource. Some examples of permissions are "execute" permission for an HTTP Servlet or JSP, and "reserve" permission for a JDBC connection pool.

In the default WebLogic Server configuration, ACLs are defined in the properties file and read by WebLogic Server at startup time. Dynamic ACLs are possible using a custom realm, as described in the next section.

ACLs are used to regulate and limit access to the majority of WebLogic Server services. Each of the following services can be protected by ACLs:

- WebLogic Events
- HTTP Servlets
- JavaServer Pages

- JDBC connection pools
- JNDI
- Workspaces
- WebLogic ZAC
- Enterprise JavaBeans
- JMS Queues and Topics

Defining an ACL

ACLs are defined in the `weblogic.properties` file with `weblogic.allow` properties. The format for this property is:

```
weblogic.allow.permission.resource={user|group}[,user|group]...
```

A *permission* is a resource-dependent string, such as "execute" for servlets or "lookup" for JNDI. See the complete list of permissions in *Using WebLogic Realms and ACLs* at *http://www.weblogic.com/docs51/classdocs/API_acl.html#setup*.

A *resource* is the name of a protected object or service in WebLogic Server.

For example, the "classes" servlet is registered and protected with these properties:

```
weblogic.httpd.register.classes=weblogic.servlet.ClasspathServlet
weblogic.allow.execute.weblogic.servlet.classes=everyone
```

The name of the servlet is "classes," as defined by the `weblogic.httpd.register` property. The ACL for the servlet grants the "execute" permission to the "everyone" group.

See *Using WebLogic Realms and ACLs* at *http://www.weblogic.com/docs51/classdocs/API_acl.html#setup* for examples of ACLs for each type of protectable WebLogic Server resource.

Security Realms

A security realm organizes security information and defines its range of operations. A WebLogic realm combines users, groups, permissions, and ACLs. It determines how a user is authenticated and retrieves users, groups, and ACLs for protected resources.

When a user attempts to access a given WebLogic Server resource, the server calls into the realm to:

- authenticate the user
- retrieve the ACL for the requested resource
- test whether the authenticated user has the needed permission on the resource

This abstraction of users and groups into realms allows you to substitute an enterprise-wide security data store for the security-related properties defined in the WebLogic Server properties file.

WebLogic includes a number of default realm extensions providing out-of-the-box security by linking to standard authentication and authorization stores, such as various LDAP servers and the Windows NT security domain. A developer can also build a custom security realm by using and extending Java classes provided with WebLogic Server. To help with this task, WebLogic Server includes an RDBMS realm example.

For example, you might prefer to use the execute permissions associated with a UNIX file system for a set of servlets stored on the UNIX computer. To enable this, WebLogic Server can defer authentication of users to UNIX and reflect access control to UNIX files in Java.

It is often necessary for security policies to be changed dynamically as new users are added and situations change. WebLogic Server can dynamically update users and groups by leveraging an external centralized security database or service such as LDAP or Kerberos.

Additional Security Considerations

WebLogic Server inherits security features from its Java environment, and from its operating system and network environments.

The Java language and runtime environment provide several built-in security features. The JVM verifies the bytecodes in a Java class file to ensure that they follow established safety rules before they are executed. This protects the JVM against class files generated by non-compliant Java compilers. The Java class loader ensures that Java class files it loads do not attempt to override existing classes. This prevents an attempt to circumvent other security measures provided by the Java runtime environment.

Java 1.2 adds a security policy feature that makes it possible to grant specific permissions to Java class files based on their point of origin. For example, you can prevent servlets loaded from a specified directory from performing any disk I/O operations.

The JVM protects itself from corruption by common programming logic errors. Such errors are caught and reported in the WebLogic Server log file. In most cases, the thread executing the errant code is unable to complete its task, but the WebLogic Server is not otherwise compromised.

The operating system on the computer running WebLogic Server can be, and should be, configured to protect the WebLogic Server installation. On a UNIX computer, you should create a WebLogic Server user and run WebLogic Server as that user. All files and directories in the WebLogic Server installation directory should belong to the WebLogic Server user and that user should have exclusive read and write access on its files and directories.

WebLogic Server also relies on the operating system to ensure the integrity of the file system it depends upon. Regular backups and other measures, such as mirrored disks, can ensure fast recovery from hardware failures.

The network environment may contain a variety of security enhancements including firewalls and proxy servers that restrict access to WebLogic Server outside of the firewall. WebLogic Server can be configured to work cooperatively with these facilities. Java clients can be permitted to tunnel to WebLogic Server over HTTP, which allows them to establish connections with WebLogic Server through a firewall or proxy.

WebLogic Server depends upon backend services, especially database servers, to provide their own security and recovery facilities. WebLogic Server uses a database to store persistent objects, including Enterprise JavaBeans and JMS messages. Therefore, database integrity is essential to WebLogic Server applications that use these features.

Administrative Utilities

The WebLogic Server distribution includes many Java utility programs that are useful for administering WebLogic Server, verifying your configuration, or diagnosing problems with your configuration. This section introduces some of these utilities.

For complete documentation on these and other WebLogic utilities, see *Using the WebLogic Utilities* at *http://www.weblogic.com/docs51/techstart/utils.html* and *Running and Maintaining WebLogic Server* at *http://www.weblogic.com/docs51/admindocs/weblogicserver.html*.

General Information Utilities

`utils.system`

Displays basic information about the Java environment, including the Java version and vendor, classpath, and operating system version.

`utils.getProperty`
: Displays *all* of the Java System properties, including the properties displayed by `utils.system` above.

`utils.myip`
: Uses the `java.net` package to display a list of the IP addresses for your computer. If you get a LicenseException error when WebLogic Server starts, use `utils.myip` to compare the IP addresses for your computer with the IP addresses specified in your WebLogic license.

Database-related Utilities

Setting up multitier database connections and JDBC connection pools requires setting up a JDBC driver and configuring two two-tier connections. The `utils.dbping` and `utils.t3dbping` utilities are useful to isolate and test the connections individually.

`utils.dbping`
: Tests a two-tier JDBC connection—a connection where a Java client connects to a database directly. You specify the JDBC driver and database connection information on the command line.You can use this utility on the computer running WebLogic Server to test a WebLogic Server JDBC connection.

`utils.t3dbping`
: Tests a three-tier database connection via WebLogic Server. `utils.t3dbping` loads the WebLogic multitier pure-Java JDBC driver and, using your command line arguments, loads a vendor-specific JDBC driver in WebLogic Server, and then establishes a three-tier database connection.

`utils.Schema`
: Executes a DDL (Data Definition Language) script in a database. `utils.Schema` establishes a JDBC connection for a database and executes the SQL commands in a file you specify. You can use this utility to create or recreate JMS system tables in a database, or rebuild the Demo database used by the WebLogic Server examples.

weblogic.Admin Class

The `weblogic.Admin` class is a Java command line application that understands several WebLogic Server administration commands. `weblogic.Admin` has at least two arguments—the WebLogic Server URL and the command name—and may require additional arguments, depending on the command you choose.

`PING`

Use `PING` to quickly check that WebLogic Server is listening on the computer and port you expect. You can add two additional parameters to specify the number of pings and the number of bytes to send on each ping. The command reports the total elapsed time and the average time per ping.

`CONNECT`

`CONNECT` is similar to `PING`, except that it actually connects to WebLogic Server and then disconnects. You can add an additional parameter to specify a number of times to connect/disconnect. The command reports the total elapsed time and the average time per connection.

`LOCK`

Disables WebLogic Server logins. This command requires the “system” username and password. You can add a message to the command, which is displayed as part of a SecurityException when connection requests are rejected.

`UNLOCK`

Re-enables WebLogic Server logins. This command requires the “system” username and password.

`SHUTDOWN`

`SHUTDOWN` shuts down WebLogic Server in an orderly fashion, first locking the server to disable new connections, and then stopping threads before exiting the JVM. This command requires the “system” username and password. You can add arguments to specify the number of seconds to delay before beginning the shutdown operation and a message to display when connections are rejected during the waiting period.

`CANCEL_SHUTDOWN`

If you set a delay on a `SHUTDOWN` request, you can cancel the shutdown with this command. You must use the `UNLOCK` command to re-enable connections after you cancel the shutdown request. Requires the “system” username and password.

`LICENSES`

If you are unable to use some WebLogic Server features and you see LicenseException messages in the WebLogic Server log, you can use the `LICENSES` command to check your licenses for WebLogic products.

Each license permits you to use a WebLogic product for a specified period of time on a specified number of computers with specified IP addresses. During startup, WebLogic Server searches its classpath for a `Licenses.xml` file or `WebLogicLicense.class` file and reads licenses from them. (The

`WebLogicLicense.class` file is a deprecated license format, but WebLogic Server still honors any unexpired licenses it finds in the file.)

With the `LICENSES` command, you can verify that WebLogic Server found your licenses and you can verify the expiration date, IP addresses, and number of units.

`VERSION`
: Displays the WebLogic Server version string.

`CREATE_POOL, DESTROY_POOL, ENABLE_POOL, DISABLE_POOL, RESET_POOL`
: These commands operate on JDBC connection pools. Use the `CREATE_POOL` and `DESTROY_POOL` commands to create or destroy JDBC connection pools from the command line. The `ENABLE_POOL` and `DISABLE_POOL` commands allow you to temporarily prevent clients from getting JDBC connections from a pool. The `RESET_POOL` command closes and re-establishes each database connection in the pool. For example, can reset the pool if connections have gone stale because of a network or database failure.

`LIST`
: The `LIST` command lists the contents of the JNDI name tree. The list includes the name each object is bound to and references to objects bound at that name. You can specify a context, such as “weblogic” or “weblogic.ejb”, to see the contents of a node in the JNDI tree.

ZAC (Zero Administration Client)

ZAC makes it easy to install and update Java applications, libraries, and applets on client computers. The ZAC Publisher is a GUI utility that you use to define and publish a client application on a WebLogic Server. You select the files that an application comprises, define dependent files or ZAC packages, and publish the package on one or more WebLogic Servers. Later, you can update the contents of a package and republish it.

You also use the ZAC publisher to create small native executable installers for each platform your application supports. You distribute the installer to clients, either in e-mail, an ftp link on a web page, on a network drive, or any other method you choose.

Clients get the installer for the package they want to install and run it to begin the installation. The installer contacts WebLogic Server and determines which files must be transferred to the client computer and transfers just those files.

Applications published with ZAC can use the ZAC API to automatically update themselves.

See *Publishing with WebLogic ZAC* at *http://www.weblogic.com/docs51/admindocs/zac.html* for help creating and publishing ZAC packages. See *Using WebLogic ZAC* at *http://www.weblogic.com/docs51/classdocs/API_zac.html* for more about using the ZAC API in your Java applications.

Performance Packs

A performance pack is a BEA-supplied native library that uses operating system calls to manage multiplexed I/O for WebLogic Server socket connections.

Operating systems provide system-level calls that monitor a set of file descriptors for I/O activity on behalf of the caller. The standard Java networking package does not provide access to these system calls—each WebLogic Server connection is represented by a Java `Socket` object that must be managed independently within WebLogic Server. Using a performance pack, WebLogic Server benefits from the more efficient multiplexing services provided by the operating system, resulting in a significant performance improvement.

Check the *Platform support page* at *http://www.weblogic.com/docs51/platforms/index.html* to see if a performance pack is available for your platform. To use a performance pack, you must use a JVM with native threads.

The performance pack is enabled by default for Windows NT. To enable the performance pack for other supported platforms, just add this property to your `weblogic.properties` file:

```
weblogic.system.nativeIO.enable=true
```

WebLogic Server Clusters

A WebLogic Server cluster provides scalability and availability by deploying multiple, cooperating WebLogic Servers behind a single network name.

A cluster promotes scalability by allowing you to add WebLogic Servers to handle increasing numbers of clients. A cluster provides availability by providing automatic failover when a WebLogic Server fails or becomes unavailable for some reason. For example, if you take a WebLogic Server out of service for maintenance, other WebLogic Servers in the cluster pick up the server's load.

Cluster Architecture

WebLogic Servers participating in a cluster can execute on heterogeneous hardware and operating systems. Each server has its own memory and disk, although a shared network disk makes it much easier to set up and maintain a cluster.

To participate in a cluster, a WebLogic Server must be accessible at a unique, static IP address, and it must be accessible by IP multicast. This means that all WebLogic Servers participating in a cluster must be on the same LAN, since multicast is not available on the Internet.

Initial load-balancing and failover are accomplished using DNS. With DNS, you map a cluster name to every WebLogic Server IP address so that the cluster appears to clients as a single network host. When multiple addresses are mapped to the same name, a DNS lookup on the name returns the entire list of addresses. If you use the standard `bind` service for DNS, the addresses on the list are rotated on each request so that consecutive requests are directed to the participating WebLogic Servers in a round-robin fashion. You can substitute software or hardware for `bind` to implement more sophisticated load-balancing algorithms.

If the WebLogic Server at an address fails to respond, the client (web browser or Java application) tries the next address on the list, providing transparent failover.

WebLogic Server clustering differs for HTTP requests and server objects accessed through JNDI, such Enterprise JavaBeans and RMI classes. For HTTP requests, the initial DNS round-robin algorithm determines which WebLogic Server will service the HTTP request. For JNDI objects, the initial DNS lookup selects a WebLogic Server to handle JNDI naming services. But when you use that naming service to look up an object, a different WebLogic Server in the cluster is chosen for the object, based on the capabilities of individual servers in the cluster and the characteristics of the object. EJB and RMI support pluggable load-balancing algorithms that can override a cluster-wide load-balancing algorithm set in the cluster's properties file.

HTTP Clustering

Servlet session data can be replicated among servers in the cluster using either JDBC persistence or in-memory replication. Session replication allows any server in the cluster to handle any request on a servlet without losing a client's session context. With JDBC persistence, sessions are stored in a database or file accessible by all servers in the cluster. With in-memory replication, which is recommended for best performance, session data is exchanged among WebLogic Servers over the network.

If you put all HTML and servlet files in a directory on a shared file system, every WebLogic Server can serve HTTP requests from the same directory. Otherwise, you must mirror the files to each WebLogic Server's directory.

RMI and EJB Clustering

RMI and EJB objects rely upon a single cluster-wide JNDI naming tree. Clustered services advertise on the cluster-wide JNDI name tree, which is replicated across the cluster so there is no single point of failure. To offer a clustered service, a WebLogic Server advertises a provider at a node in the replicated naming tree. Each server adds a stub for that provider to a service pool stored in its copy of the naming tree.

Both RMI and EJB send remote stubs to their clients. The stubs manage communications between the client and the associated server-side objects. Stubs delivered to the client can be clusterable or nonclusterable. Consecutive calls on clusterable stubs may be handled by any WebLogic Server in the cluster, depending on load-balancing and failover requirements when a call is made. With a non-clusterable stub, all calls on a stub are handled by the WebLogic Server that initially delivered the stub to the client. Clusterable stubs can failover transparently; nonclusterable stubs cannot.

Some RMI and EBJ objects must not be clustered because they have state in the server-side object. The client must always interact with the same instance of the object on the same WebLogic Server. In this case, you can take advantage of load-balancing and failover up to the point where a stateful object is instantiated. For example, an EJB home interface has no state and is clusterable. You can look up an EJB home and retrieve a clusterable stub. But when you use a method on the home interface to create or look up a stateful session bean or entity bean, you get a non-clusterable stub. All calls you make on the bean are handled on the WebLogic Server where the bean was instantiated. A remote object with nonclusterable stubs is said to be "pinned" to the WebLogic Server where it lives.

To take advantage of clustering with stateful objects, you can wrap operations in transactions and then, if the host WebLogic Server fails, look up the object again and reapply the transaction. For example, if the WebLogic Server hosting an entity bean instance fails, you can use the same home interface to look up or create the bean again. The home interface has a clusterable stub and failover capability, so it will automatically choose another WebLogic Server to host the new bean instance.

Unlike EJB, RMI does not define categories of stateful and stateless objects, so you must intentionally "pin" a stateful RMI object to its server. One way to do this is to emulate the EJB home interface by looking up a clusterable factory method in the replicated JNDI tree. Calls on the factory method can execute on any server in the cluster, but objects that you create with the factory are pinned to their server.

You can also go directly to a specific WebLogic Server in a cluster, bypassing the cluster-wide JNDI name tree, to look up an EJB or RMI class. This ensures that your remote objects are pinned to the selected server, but does not take advantage of load balancing.

JDBC clustering

WebLogic JDBC connection pools can be registered in the replicated naming tree so that applications can advantage of load balancing and failover provided by a WebLogic Server cluster.

In the usual case, each WebLogic Server in the cluster is configured with a JDBC connection pool that accesses the same back-end DBMS instance. Each WebLogic Server then has a pool of connections to the same database. A connection pool is registered in the naming tree by defining a DataSource property in the `weblogic.properties` file.

You could use different database instances for each WebLogic Server if your application does not require a shared database, or if the DBMS has replication or mirroring services that make it possible for the various database instances to behave as a single instance.

A client uses the WebLogic JDBC/RMI JDBC driver to get a database connection from a cluster. This driver, new in WebLogic Server version 5.1, makes it possible to get a database connection from a pool using JNDI. When a client looks up the DataSource name for a connection pool, the load-balancing algorithm selects the WebLogic Server that will serve the request. Since a JDBC connection maintains state on WebLogic Server, a JDBC connection is pinned to the WebLogic Server that is selected to serve a client's request.

See *Using WebLogic JDBC/RMI and Clustered JDBC* at *http://www.weblogic.com/docs51/classdocs/JDBC_RMI.html* for information about clustered JDBC.

Non-clustered Services in a WebLogic Server Cluster

WebLogic JMS is not a clustered service in the current WebLogic Server release. If you use JMS, select one WebLogic Server in the cluster to provide JMS services for the cluster and configure JMS in its per-server properties file. More importantly, do *not* configure any other WebLogic Server for JMS. When the WebLogic Server configured for JMS starts up, its JMS ConnectionFactories, Queues, and Topics, are added to the cluster-wide JNDI tree. Any client that looks up a JMS object is then directed to the WebLogic Server that is configured for JMS.

More about WebLogic Clusters

To learn more about WebLogic clusters, refer to these documents:

See *Setting up a WebLogic cluster* at *http://www.weblogic.com/docs51/admindocs/cluster.html* for help setting up clustering.

See *Using WebLogic clusters* at *http://www.weblogic.com/docs51/classdocs/API_cluster.html* for help developing clustered WebLogic Server applications.

See *Using WebLogic JDBC/RMI and Clustered JDBC* at *http://www.weblogic.com/docs51/classdocs/JDBC_RMI.html* for help with setting up and using clustered JDBC.

CHAPTER

3 WebLogic Server Developer APIs

WebLogic Server provides developers with a comprehensive set of Java APIs (Application Programming Interfaces) and a powerful deployment environment.

BEA is an enthusiastic supporter and participant in Sun's Java Community Process, the open specification development process that has produced several standard Java technologies implemented in WebLogic Server. BEA has committed to ongoing full support for these Java standards, including Sun's J2EE (Java 2 Platform, Enterprise Edition) announced in June 1999. WebLogic Server offers the most complete and mature J2EE support available.

The next several chapters introduce the WebLogic Server development APIs in a tutorial format. Our online documentation includes a detailed *Developers Guide* for each of the APIs described here. Javadoc references for *WebLogic APIs* are also available online at *http://www.weblogic.com/docs51/classdocs/javadocs/index.html*. The *Javadoc references for Sun APIs* are available on the Sun web site at *http://www.javasoft.com/products/index.html*.

Developing Applications for WebLogic Server

Typical WebLogic Server applications are Java classes running on the server, using Servlets and JSP pages to interact with web browser clients. Business logic executes on WebLogic Server and only HTTP passes over the network. Servlets and JSP pages can call other Java classes and WebLogic Server facilities to accomplish their work. Except for HTML, the code you write is intended to run on WebLogic Server.

If you write a Java client instead of a web-based client, it is good practice to keep your business logic on WebLogic Server and concentrate on user interface and presentation in the client. RMI provides a nearly transparent distributed programming framework, so some application code that runs on WebLogic Server could also run in a Java client. Executing classes remotely adds overhead and has performance implications, so it is best to avoid programming business logic in Java clients.

About the Examples in this Document

You can find the source code for the examples in the following chapters in the *examples/intro* directory of your WebLogic Server installation.

To access and run the examples, first install WebLogic Server, following instructions in *Installing and Setting up WebLogic Server* at *http://www.weblogic.com/docs51/install/index.html*. The WebLogic Server distribution includes scripts to set up your environment for running WebLogic Server and building examples to use with WebLogic Server. See *Setting up Your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html* to learn how to edit and execute these scripts before you try the examples in this chapter.

The example code presented in this document assumes that you are using a Java Developers Kit that is compatible with the JavaSoft JDK. The Java compiler is named **javac**, the JVM is named **java**, and options are provided to these utilities as defined by the JavaSoft JDK. You can use another Java development environment if it is certified for use with WebLogic Server. (See our *Platform Support Page* at *http://www.weblogic.com/docs51/platforms/index.html* for a list of certified JDKs.) *If you use a non-JavaSoft JDK, you may have to translate the instructions in this chapter to work with your tools.*

The Enterprise JavaBean example requires the **jar** tool to package the bean for deployment on WebLogic Server. The Microsoft Developers Kit does not provide a **jar** command. To try this example, and the JMS example that uses the bean, you need the JavaSoft JDK.

What's Next?

The remaining chapters provide a tutorial for developing WebLogic Server applications. Each chapter describes a development API and gives an example, including instructions for compiling the code, configuring WebLogic Server for the example, and executing the example:

- Servlets
- JSP (JavaServer Pages)
- JDBC (Java Database Connectivity)
- EJB (Enterprise JavaBeans)
- JMS (Java Message Service)

CHAPTER

4 Servlets

A servlet is a server-side Java class that can be invoked and executed, usually on behalf of a client. Do not confuse servlets with applets; a servlet does its work on the server, but an applet does its work on the client. Several WebLogic Server services are implemented using servlets.

A servlet has input and output objects that represent a request and response, respectively. When a client requests a servlet, WebLogic Server loads the servlet, calls its `init()` method, and passes the request and response objects to a service method.

WebLogic Server supports the `javax.servlet.GenericServlet` and `javax.servlet.http.HttpServlet` interfaces. An `HttpServlet` extends the `GenericServlet` interface, adding support for the HTTP protocol.

The HTTP protocol concerns client requests and server responses. Clients are usually web browsers and servers are usually web servers, including servlet engines like WebLogic Server. Other web servers can be configured to redirect their servlet requests to WebLogic Server.

HTTP servlets (and JSP, which is based on HTTP servlets) are the components that represent the server-side presentation logic in e-commerce applications. This section concentrates on HTTP servlets.

HTTP servlets extend the `javax.servlet.http.HttpServlet` class, which allows them to run in any compliant servlet engine. The work performed by HTTP servlets can be compared to that of CGI scripts, but HTTP servlets are easier to write, faster than running an external script, and they have access to other WebLogic Server services.

The `javax.servlet.http` package provides the Java interfaces that allow your code to participate in the HTTP protocol. An HTTP servlet is called with an `HttpServletRequest` object, which contains data received by the server from the client, and an `HttpServletResponse` object, where the servlet can write data to return to the client.

A servlet can extract standard headers from the client request. When clients enter data in HTML forms, the data is transferred to the servlet as a set of parameters in the HTTP stream. The servlet can read the parameters by calling the `HttpServletRequest` object's `getParameterValues()` method.

To respond to an HTTP request, a servlet uses the `HttpServletResponse` object. The methods on this object let the servlet take such actions as setting headers in the HTTP response, adding a cookie to a response, or returning an error. The `HttpServletResponse` also provides a `PrintWriter` to the servlet, which the servlet uses to write HTML. Since the HTML is generated for each request to the servlet, it can be highly dynamic.

Many web applications require a coordinated series of requests and responses so they can get input from the client and return the correct response. To link HTTP requests and retain data over a series of calls, many web applications make use of URL parameters and browser cookies. With URL parameters, data is appended to the URL a client uses to make a request. Cookies allow a server to store an object in the client's web browser for later access, if the browser user permits it.

The `javax.servlet.http.HttpSession` interface provides a convenient method for managing a series of HTTP events in a servlet. This interface allows a servlet to create a session object in the servlet and store data in it. The session remains available to the servlet when the same client returns. WebLogic Server ties the session data it maintains to a particular browser session by setting a cookie in the browser. If the user has disallowed cookies, the servlet can still use sessions by coding the session ID in the URL. See *Using WebLogic HTTP Servlets* at *http://www.weblogic.com/docs51/classdocs/API_servlet.html* for information on URL rewriting.

WebLogic Server can be configured as a fully functional web server that delivers any arbitrary MIME type to HTTP clients. For more information on configuring WebLogic Server HTTP services, see *Setting up WebLogic as an HTTP Server* at *http://www.weblogic.com/docs51/admindocs/http.html*.

Creating an HTTP Servlet

A servlet class handles HTTP requests by overriding methods in `javax.servlet.http.HttpServlet`. A client can make an HTTP request to a servlet, specifying one of these service methods: `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, or `TRACE`. The `javax.servlet.http.HttpServlet` class provides a default empty implementation for each of these methods. To create a servlet, you just override the methods you want to implement.

For example, when a user enters the servlet's URL into a browser, the browser sends a `GET` request to the servlet. The servlet engine sends the request to the servlet's `service()` method, which routes the `GET` request to the `doGet()` method. The default `doGet()` implementation returns an error 400, `BAD_REQUEST`, which is the correct response for a servlet that does not handle a `GET` request.

See the *Javadocs* for `javax.servlet.HttpServlet` on the Sun web site for a description of each of the service methods and their default implementations.

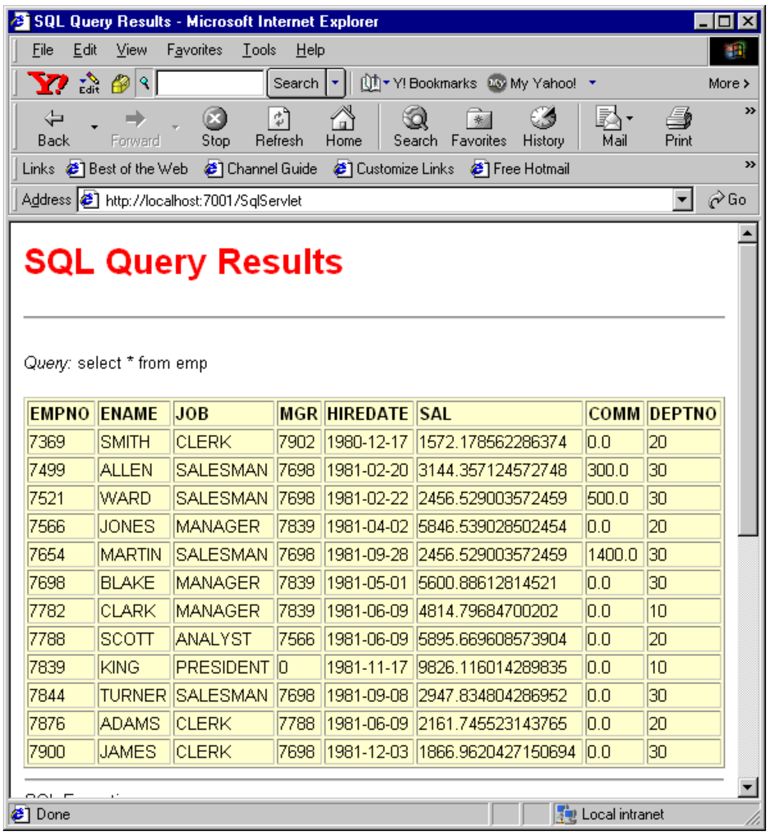
In HTTP servlets, it is common to override the `doGet()` and `doPost()` methods, less common to override the other methods. With generic servlets, all requests are passed to the `service()` method.

A servlet can override the `init()` method, which is called whenever the servlet is loaded or reloaded. It can perform any initialization work that needs to be done before it responds to client requests, but it must first call the `init()` method in the parent class. The `init()` method is commonly used to retrieve configuration parameters for the servlet which, in WebLogic Server, are set in the `weblogic.properties` file. You can override the `destroy()` method to perform any clean up required when the servlet exits.

The HTTP service methods, such as `service()`, `doGet()` and `doPost()`, are each called with an `HttpServletRequest` and an `HttpServletResponse`. The servlet gets information about the client's request from the `HttpServletRequest` and writes the response into the `HttpServletResponse`.

SqlServlet example

The `SqlServlet.java` example is a servlet that gets an SQL statement from a browser form, executes the statement using a connection from a WebLogic Server connection pool, and constructs an HTML page containing the query results. Here is an example of the output from this servlet:



SQL Query Results

Query: select * from emp

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	1572.178562286374	0.0	20
7499	ALLEN	SALESMAN	7698	1981-02-20	3144.357124572748	300.0	30
7521	WARD	SALESMAN	7698	1981-02-22	2456.529003572459	500.0	30
7566	JONES	MANAGER	7839	1981-04-02	5846.539028502454	0.0	20
7654	MARTIN	SALESMAN	7698	1981-09-28	2456.529003572459	1400.0	30
7698	BLAKE	MANAGER	7839	1981-05-01	5600.88612814521	0.0	30
7782	CLARK	MANAGER	7839	1981-06-09	4814.79684700202	0.0	10
7788	SCOTT	ANALYST	7566	1981-06-09	5895.669608573904	0.0	20
7839	KING	PRESIDENT	0	1981-11-17	9826.116014289835	0.0	10
7844	TURNER	SALESMAN	7698	1981-09-08	2947.834804286952	0.0	30
7876	ADAMS	CLERK	7788	1981-06-09	2161.745523143765	0.0	20
7900	JAMES	CLERK	7698	1981-12-03	1866.9620427150694	0.0	30

The `SqlServlet` example demonstrates the following:

- An `init()` method that retrieves an `initArg` that is set in the `weblogic.properties` file. The argument specifies the DataSource the servlet will use.
- A `doGet()` method that sends an HTML form to the browser.
- A `doPost()` method that retrieves the SQL statement entered in the form and calls some additional methods to process the query and format the results.

init() method

The `init()` method of a servlet does whatever initialization work is required when WebLogic Server loads the servlet. The default `init()` method does all of the initial work that WebLogic Server requires, so you do not need to override it unless you have special initialization requirements. If you do override `init()`, first call `super.init()` so that the default initialization actions are done first.

The `SqlServlet` `init()` method gets the name of the DataSource, which provides JNDI access to the JDBC connection pool it is to use. Setting the DataSource name as an argument makes it easy to redeploy the servlet with a different database.

Here is the beginning of the `SqlServlet.java` example, including the `init()` method:

Listing 4-1 SqlServlet.java

```
package examples.intro;
/*
 * @author Copyright (c) 1999-2000. BEA Systems, Inc.
 * All rights reserved.
 */

import java.io.*;
import java.sql.*;
import java.util.Hashtable;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.DataSource;
import weblogic.common.*;


/**
 * This example servlet takes an SQL query from a form and
 * executes it using a connection borrowed from a connection
 * pool in WebLogic Server.
 *
 * The name of the connection pool is specified in an initArg
 * for the servlet in the <tt>weblogic.properties</tt> file.
 */

public class SqlServlet extends HttpServlet
{
  String connPool;

  /**
   * The init method performs any tasks that need to be done when a
   * servlet first starts up.  For this servlet, there is one task,
   * which is to look up the name of the DataSource to use for SQL
   * queries. The DataSource is set in the servlet’s initArgs
   * property in the <tt>weblogic.properties</tt> file.
   */
  public void init(ServletConfig config)
    throws ServletException
  {
    super.init(config);
    connPool = getInitParameter("dataSource");
    if (connPool == null) {
      System.err.println("This servlet requires a " +
                         " dataSource initArg." );
    }
  }
```

`SqlServlet` imports `javax.servlet.*` and `javax.servlet.http.*` and extends `HttpServlet`. The `java.sql` and `weblogic.common` packages are imported because this servlet uses JDBC and WebLogic JDBC connection pools. The `java.io` package is imported for the I/O methods of the `PrintWriter`, which is a stream object.

Our purpose in overriding the `init()` method is to retrieve the name of the connection pool. In the `weblogic.properties` file, this servlet is registered with properties similar to these:

```
weblogic.http.register.SqlServlet=\
     examples.intro.SqlServlet

weblogic.http.initArgs.SqlServlet=\
     dataSource=demoDataSource
```

The first property registers the servlet with the name SqlServlet. The second passes a single argument, `dataSource`, which is set to the name of the DataSource for the a JDBC connection pool configured in the WebLogic Server.

The `init()` method retrieves the argument by calling `getInitParameter("dataSource")`.

doGet() method

The URL for a servlet in WebLogic Server is the URL of the WebLogic Server followed by the registered name of the servlet. The URL for the `SqlServlet` servlet is:

`http://`*`host`*`:`*`port`*`/SqlServlet`

where *`host`* is the name of the computer running WebLogic Server and *`port`* is the port number where WebLogic Server is listening for connections. If a user requests this

URL in a browser, WebLogic Server passes the request to the `doGet()` method of the `SqlServlet` servlet.

`doGet()`, and the `putForm()` method it calls, construct an HTML form to get an SQL query from the client:

Listing 4-2 SqlServlet.java doGet() method

```
 /**
   * The doGet() method is called when a Servlet is first
   * requested. This implement prints a query form to get
   * the query from the browser. Once a client has this
   * form, all of their subsequent calls to this servlet
   * should be through the doPost() method.
   */
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException
  {
    res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    pw.println("<html><head><title>SQL Query Servlet");
    pw.println("</title></head>");
    pw.println("<body bgcolor=#ffffff><font face=\"Helvetica\">");
    pw.println("<h1><font color=\"#FF0000\">");
    pw.println("SQL Query Servlet</font></h1>");
    pw.println("<hr><p>");
    putForm(req, pw, "");
    pw.println("</body></html>");
    pw.close();
  }

 /**
   * putForm() writes an HTML form to the response to collect
   * the sqlStatement parameter.
   */
  public void putForm(HttpServletRequest req,
                      PrintWriter pw, String query)
    throws IOException
  {

    pw.println("<form action=\"" +
               req.getRequestURI() +
               "\" method=post>");
    pw.println("Enter an SQL query:");
    pw.println("<center><p>");
    pw.println("<textarea name=\"sqlStatement\" " +
             "rows=\"8\" cols=\"64\" wrap=hard>" +
             query + "</textarea>");
  pw.println("<p><input type=\"submit\" " +
             "name=\"sendSql\" value=\"Send SQL\">");
  pw.println("</center>");
  pw.println("</form>");
}
```

doGet() sets the MIME type of the page it is generating to "text/html", and then gets a PrintWriter from the HttpServletResponse. Then it writes out standard HTML through the PrintWriter. The HTML code for the form is in a separate method so that it can be reused by the doPost() method.

doGet() and putForm() have no dynamic HTML content. They could be implemented as static HTML pages. The <form> tag addresses the contents of the form to the SqlServlet servlet. The action attribute gives the URL of the servlet, and the method attribute specifies the HTTP POST method. The URL in the example code is relative to the servlet. If you want to send the data to another WebLogic Server or to another servlet, you can give the complete URL in the action attribute.

doPost() method

doPost() is called when a client clicks the “Send SQL” button on the form. The name attributes in the form element tags become the parameter names used to transport the form input in the HTTP request. In a doPost() method, parameters are retrieved by calling getParameterValues() on the HttpServletRequest. getParameterValues() returns an array of Strings, or null. The SqlServlet servlet is only interested in the first element of the array returned by the sqlStatement parameter, which contains the SQL statement the user entered:

Listing 4-3 SqlServlet.java doPost() method

```
/**
  * This method handles a POST request from a client. It
  * gets the SqlStatement parameter from the HttpServletRequest
 * object, executes the SQL using a connection from the connection
  * pool, and generates an HTML page with the query results.
  */
 public void doPost(HttpServletRequest req,
                    HttpServletResponse res)
```

```
    throws ServletException, IOException
  {
    // first, set the "content type" header of the response
    res.setContentType("text/html");

    //Get the response’s PrintWriter to return text to the client.
    PrintWriter pw = res.getWriter();
    pw.println("<html><head><title>" +
               "SQL Query Results</title></head>");
    pw.println("<body bgcolor=#ffffff><font face=\"Helvetica\">");
    pw.println("<h1><font color=\"#FF0000\">");
    pw.println("SQL Query Results</font></h1>");
    pw.println("<hr><p>");

    try {
      String theQuery = req.getParameterValues("sqlStatement")[0];
      pw.println("<p><i>Query:</i> " + theQuery + "<p>");

      doQuery(theQuery, pw); // all the JDBC is in this method
      pw.println("<p><hr>");
      putForm(req, pw, theQuery); // ask for another SQL statement
      pw.println("<p><hr></body></html>");
    }
    catch (Exception e) {
      pw.println("<p><hr>A problem occurred:");
      pw.println("<pre>" + e.getMessage() + "</pre>");
    }
    finally {
      pw.close();
    }
  }
```

`doPost()` writes the HTML for the response, but it calls `doQuery()` to handle all of the JDBC and write out the HTML-formatted query results, and `putForm()` to add the query form following the results.

Support methods

`doPost()` calls `doQuery()` to handle all of the JDBC processing for the query. The query could be any text—`doQuery()` just sends it on to the JDBC driver. The result of the query could be a syntax error or some other exception, a row update count, or a set of rows. Some SQL commands, such as stored procedures, could return multiple result sets. The try block in `doQuery()` shows how to process an arbitrary `ResultSet`.

Listing 4-4 SqlServlet.java doQuery() method

```
/**
   * This method executes the SQL query.
   */
  public void doQuery(String theQuery, PrintWriter pw)
    throws Exception
  {
    // set up the JDBC connection
    Context ctx = null;
    Hashtable ht = new Hashtable();
    Connection conn = null;
    ht.put(Context.INITIAL_CONTEXT_FACTORY,
           "weblogic.jndi.WLInitialContextFactory");
    try {
      ctx = new InitialContext(ht);
      javax.sql.DataSource ds =
        (javax.sql.DataSource) ctx.lookup("demoDataSource");
      conn = ds.getConnection();
      conn.setAutoCommit(true);
      Statement s = conn.createStatement();
      ResultSet rs = null;
      s.execute(theQuery);

      // loop through the result sets
      while (true) {
        rs = s.getResultSet();
        int rowcount = s.getUpdateCount();
        if (rowcount != -1) {
          pw.println("Update count = " + rowcount);
        }
        if (rs == null && rowcount == -1)
          break;
        if (rs != null)
          // call another method to construct a table
          // from the ResultSet
          printResults(pw, rs);
        s.getMoreResults();
      }
    }
    catch (NamingException ne) {
      pw.print("<p><hr>JNDI Exception: ");
      pw.print("<pre>" + ne.getMessage() + "</pre>");
    }
    catch (SQLException se) {
          pw.print("<p><hr>SQL Exception: ");
          pw.println("<pre>" + se.getMessage() + "</pre>");
    }
    finally {
      try { conn.close(); } catch (Exception e) {};
      try { ctx.close(); } catch (Exception e) {};
    }
  }
```

The `printResults()` method outputs a `ResultSet` in a formatted HTML table. It gets the metadata from the `ResultSet`, which provides access to the column names. It loops through the result rows, adding an HTML row for each.

Listing 4-5 SqlServlet.java printResults() method

```
  /** This method is called from doPost to format the
   * query results as an HTML table.
   */
  static void printResults(PrintWriter pw, ResultSet rs)
    throws IOException, SQLException
  {
    int columns = 0;
    ResultSetMetaData md = null;

    pw.println("<table border=1 bgcolor=\"#FFFFCC\" " +
               "width=\"100%\">");
    md = rs.getMetaData();
    columns = md.getColumnCount();
    pw.println("<tr>");
    for (int i = 0 ; i < columns; i++ ) {
      pw.print("  <th align=left>");
      String label = md.getColumnLabel(i+1);
      pw.print( label == null?"<br>" : label ) ;
      pw.println("</th>");
    }
    pw.println("</tr>");
    while (rs.next()) {
      pw.println("<tr>");
      for (int i = 0; i < columns; i++) {
        Object val = rs.getObject(i+1);
        if (val == null)
          pw.println("  <td><br></td>");
        else
          pw.println("  <td>" + val.toString().trim()
                     + "</td>");
      }
      pw.println("</tr>");
      }
      pw.println("</table>");
      rs.close();
    }
```

Running the SqlServlet example

1. Set your development environment as described in *Setting your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html*.
2. In your `weblogic.properties` file, if you have not set up a JDBC connection pool, uncomment the `weblogic.jdbc.connectionPool.demoPool` property to use the Cloudscape evaluation database.
3. Make sure there is an ACL (Access Control List) for the connection pool that includes either “guest” or “everyone”. If you are using the Cloudscape `demoPool` connection pool, for example, uncomment the property:

```
weblogic.allow.reserve.weblogic.jdbc.connectionPool.demoPool=\
       everyone
```

   This property is needed because `SqlServlet` does not specify a username and password when it gets the connection from the pool. Whenever an unauthenticated client attempts to use a WebLogic service, WebLogic Server assigns the client the “guest” user identity. If the connection pool does not have an ACL that allows “guest” to use the pool, the servlet will get an exception with a message such as: `Pool connect failed: User "guest" does not have Permission "reserve" based on ACL "weblogic.jdbc.connectionPool.demoPool"`.

4. Create a DataSource entry for the “demoPool” connection pool by adding this property:

```
weblogic.jdbc.TXDataSource.demoDataSource=demoPool
```

5. Register the servlet by adding these properties:

```
weblogic.http.register.SqlServlet=\
     examples.intro.SqlServlet

weblogic.http.initArgs.SqlServlet=\
     dataSource=demoDataSource
```

6. In your development shell, change to the directory where you copied the examples, and compile `SqlServlet.java` into the `SERVLET_CLASSES` directory:

   Windows NT:

   ```
   javac -d %SERVLET_CLASSES% SqlServlet.java
   ```

   UNIX:

   ```
   javac -d $SERVLET_CLASSES SqlServlet.java
   ```

7. Start WebLogic Server.

8. In a web browser, enter the URL for the servlet. For example, if WebLogic Server and your browser are on the same computer, enter `http://localhost:7001/SqlServlet`. Otherwise, change *localhost* to the name of the computer running WebLogic Server.

9. Enter a SQL query in the browser form and click the “Send SQL” button.

   You can enter any SQL DDL or DML statement that is valid for the target database. Since `SqlServlet` uses a JDBC `Statement` to execute the SQL, it can only handle results that can come from a `Statement`. For example, it does not handle procedure parameters.

More about Servlets

Learn more about writing HTTP Servlets in the developers guide, *Using WebLogic HTTP Servlets* at *http://www.weblogic.com/docs51/classdocs/API_servlet.html*.

You can find javadocs for the *javax.servlet.http.HttpServlet* package on the Sun website at *http://jserv.java.sun.com/products/java-server/documentation/webserver1.1 /apidoc/javax.servlet.http.HttpServlet.html*.

To configure WebLogic Server as an HTTP server, see the administration document *Setting up WebLogic as an HTTP Server* at *http://www.weblogic.com/docs51/admindocs/http.html*.

5 JSP (JavaServer Pages)

JSP (JavaServer Pages) is a standard for combining Java and HTML to provide dynamic content in web pages. JSP is part of Sun's Java 2 Platform, Enterprise Edition. WebLogic Server 4.5 implements the JSP 1.1 specification.

JSP depends upon HTTP servlets—the JSP compiler generates the source code for an HTTP servlet from a .jsp file. The Java classes produced using the HTTP servlet and JSP are essentially equivalent. You can use either technology to create dynamic web pages.

JSP emphasizes page design with standard HTML formatting, so it is more convenient to use JSP for creating dynamic web pages than it is to write a Java HTTP servlet.

With JSP, you embed Java code in HTML using special JSP tags similar to HTML tags. You install the JSP page, which has a .jsp extension, into the WebLogic Server document root, just as you would a static HTML page. When WebLogic Server serves a JSP page to a client, it tests whether the file has been compiled. If not, it calls the WebLogic JSP compiler to compile the file into a servlet. A .jsp file is only recompiled when it has changed.

Although you can embed as much Java code as you like into a JSP page, using JavaBeans and Enterprise JavaBeans lets you concentrate on page design and presentation in the JSP page and encourages you to move application logic into reusable components. Using beans from JSP pages also makes it possible for web page designers to create and maintain dynamic web pages without having to learn Java programming.

JSP Tags

JSP defines the following tags to enclose Java code fragments and directives for the JSP compiler:

`<%! `*`declaration`*` %>`
: Contains class-level declarations of Java variables or methods. The variables and methods declared can be referenced within other JSP tags on the page.

`<% `*`scriptlet`*` %>`
: Contains a Java code fragment that executes when the JSP page executes.

`<%@ include file="`*`relativeURL`*`" %>`
: Includes the contents of a file at compile time. The JSP compiler includes the file and processes its contents.

`<%@ page `*`attributes`*` %>`
: Defines page-level attributes. Attributes are specified as *`name`*`="`*`value`*`"` pairs. Multiple attributes are separated by white space within the tag.

`<%= `*`expression`*` %>`
: Writes the value of a Java expression into the response page.

`<!-- <%= `*`expression`*` %> -->`
: Evaluates a Java expression within an HTML comment. The resulting comment is not rendered by the browser, but can be viewed in the page source.

`<%-- `*`comment`*` --%>`
: Comments the JSP source. Comments written within these tags are not included in the HTML output.

`<jsp:useBean ... />`
: Specifies a JavaBean to be used on the JSP page.

`<jsp:setProperty ... />`
: Sets JavaBean properties.

`<jsp:getProperty ... />`
: Gets JavaBean properties.

JSP tags can also be written with XML tags. See *Using WebLogic JSP* at *http://www.weblogic.com/docs51/classdocs/API_jsp.html* for a table of equivalent tags.

Implicit Variables

JSP provides a number of implicit variables that can be referenced on any JSP page. The JSP compiler adds these class variable declarations to the generated servlet code:

```
HttpServletRequest request;   // Request object
HttpServletResponse response; // Response object
HttpSession session;          // Session object for the client
ServletConfig config;         // ServletConfig for this JSP
ServletContext application;   // Servlet context
PageContext pageContext;      // Page context for this JSP
Object page;                  // Instance of the implementation
                              // class for this JSP (i.e., ’this’)
JspWriter out;                // PrintWriter to the browser
```

The `request` and `response` variables provide access to the HTTP request and response objects. A JSP page can extract information from the `request` and write information on the `response` using methods from the `javax.servlet.http` package.

The `session` variable allows WebLogic Server to save state for a client between requests on a JSP page. You must set the `"session=true"` attribute in the JSP `page` directive so that the JSP compiler includes support for sessions. Then you can save arbitrary values and retrieve them again on subsequent requests with the `getValue()` and `putValue()` methods on the session.

The `config` variable provides access to the `javax.servlet.ServletConfig` object for the JSP page. The methods on this object return initialization parameters for the page. With WebLogic Server, you define initialization parameters by setting an `initArg` property when you register a servlet in the `weblogic.properties` file. To use initialization arguments with JSP pages, you must register the JSP page as a servlet.

The `application` variable gives you access to information about the environment of the JSP page and provides methods for writing messages in the WebLogic Server log.

The `pageContext` variable provides access to a `javax.servlet.jsp.PageContext` object. You can use methods on this object to access various scoped namespaces, servlet-related objects, and common servlet-related functionality.

The `page` variable references the current instance of the JSP page. `page` is declared as a `java.lang.Object`, so you must cast it to the name of the generated servlet class name if you want to use it. An easier alternative is to just refer to the current instance of the page with the Java `this` keyword.

The `out` variable is a Java `PrintWriter` you can use to write HTML to the browser page from inside JSP tags.

A Simple JSP Example

The `Calendar.jsp` example is a simple JSP page that calls a JavaBean to display a calendar for the current month. The source for the JavaBean is in the file `CalendarBean.java`.

`CalendarBean` gets the current date and provides two get methods, `getTodayString()` and `getHtmlMonth()`. `getTodayString()` returns the current date in a String such as `"Thursday, October 14, 1999"`. `getHtmlMonth()` returns a String containing the calendar for the current month, formatted as an HTML table.

`CalendarBean` has one `set` method, `setColor()`, which is used to set the background color of the calendar.

`Calendar.jsp` shows how to specify a JavaBean, set its properties, and get its values from within a JSP page. The source of the JSP page is fairly simple, but it demonstrates how complicated web page elements can be delegated to reusable JavaBean components.

Here is the source for `Calendar.jsp`:

Listing 5-1 Calendar.jsp

```
<!doctype html public "-//w3c/dtd HTML 4.0//en">
<html>
<head><title>Calendar</title></head>

<%@ page
        info="Calendar JSP example"
        contentType="text/html"
%>

<jsp:useBean id="calendar"
       scope="page"
       class="examples.intro.CalendarBean"
/>


<jsp:setProperty name="calendar"
```

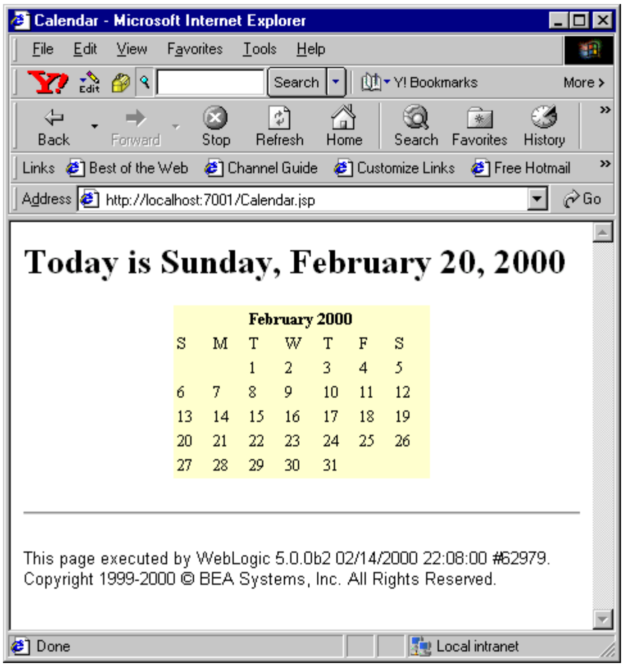
```
     property="Color" value="#FFFFCC"/>


<h1>Today is <jsp:getProperty name="calendar"
       property="TodayString"/></h1>

<p>
<center>
<jsp:getProperty name="calendar" property="HtmlMonth" />
</center>



<p>
<hr>
<font face="Helvetica">
<p>This page executed by
<%= application.getServerInfo() %>.<br>
Copyright 1999-2000 &copy; BEA Systems, Inc.
All Rights Reserved.
</body>
</html>
```

The `<jsp:useBean ... />` tag identifies the CalendarBean class and assigns it an ID (`calendar`), which is used to reference it in other JSP tags on the page. The `scope` attribute specifies that the instance of this bean is stored in the current page. Setting the `scope` to `"session"` would tell WebLogic Server to store the bean instance in the session so that any changes persist over multiple requests on the JSP page.

The `<jsp:setProperty ... />` tag sets the background color of the HTML table. The JSP compiler translates this to a call to the `CalendarBean.setColor()` method.

The `<jsp:getProperty ... />` tags are translated into calls to `CalendarBean.getTodayString()` and `CalendarBean.getHtmlCalendar()`. The results of those calls are embedded in the HTML output.

Here is an example of the output from the `Calendar.jsp` JSP page:


Calendar - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Search
Y! Bookmarks
More
Back Forward Stop Refresh Home Search Favorites History
Links Best of the Web Channel Guide Customize Links Free Hotmail
Address http://localhost:7001/Calendar.jsp
Go
Done
Local intranet

Today is Sunday, February 20, 2000

February 2000						
S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

This page executed by WebLogic 5.0.0b2 02/14/2000 22:08:00 #62979.
Copyright 1999-2000 © BEA Systems, Inc. All Rights Reserved.

Running the Calendar.jsp Example

To use `Calendar.jsp`, your WebLogic Server must be configured to serve JSP pages. Then you compile `CalendarBean.java`, copy `Calendar.jsp` into WebLogic Server's document root, and call the page in a browser. Here are the steps:

1. If you have not yet configured WebLogic Server for JSP, edit the `weblogic.properties` file in the WebLogic installation directory. Find and uncomment the `weblogic.httpd.register.*.jsp` and `weblogic.httpd.initArgs.*.jsp` properties. Edit the `compileCommand`

attribute to point to the location of your Java compiler. Edit the `workingDir` attribute so that it points to an existing directory where the JSP compiler can store its generated files.

For more help on these properties see *Setting up WebLogic for JSP* at *http://www.weblogic.com/docs51/classdocs/API_jsp.html#setup* in "Using WebLogic JSP".

2. Set up your development environment as described in *Setting your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html*.

3. Change to the directory where you copied the example files and compile the `CalendarBean.java` file.

   Windows NT:

   ```
   javac -d %SERVER_CLASSES% CalendarBean.java
   ```

   UNIX:

   ```
   javac -d $SERVER_CLASSES CalendarBean.java
   ```

4. Copy the `Calendar.jsp` file into the document root of your WebLogic Server. By default, this is the `myserver/public_html` directory in your WebLogic installation.

5. Start WebLogic Server. See *Setting up and Starting WebLogic Server* at *http://www.weblogic.com/docs50/admindocs/startserver.html* for help.

6. In a browser, enter the URL for the JSP page. For example:

   ```
   http://localhost:7001/Calendar.jsp
   ```

More about JSP

You can find more about using WebLogic JSP in the developers guide *Using WebLogic JSP* at *http://www.weblogic.com/docs51/classdocs/API_jsp.html*.

See the *Sun JSP page* at *http://www.java.sun.com/products/jsp/index.html* to find JSP specifications, tutorials, and examples.

CHAPTER

6 JDBC (Java Database Connectivity)

JDBC provides standard interfaces for accessing relational database systems from Java. To use JDBC, you need a JDBC driver for the database system you want to access. BEA provides JDBC drivers for Oracle, Microsoft SQL Server, and Informix Dynamic Server. JDBC drivers for Cloudscape and Sybase SQL Server are bundled with WebLogic Server. And you can use any standard JDBC driver with WebLogic Server.

JDBC can be employed in two-tier and multitier configurations. In a two-tier configuration, a Java client loads the JDBC driver into its own JVM. If the JDBC driver is a type 2 driver—one that employs vendor-specific native client libraries—the client must have the database vendor's client libraries installed.

In a multitier configuration, WebLogic Server loads a JDBC driver into its JVM and establishes the connection to the database. Instead of loading the database-specific JDBC driver, the client application loads a WebLogic JDBC driver, which is a pure-Java, database-independent driver. A multitier configuration is composed of two two-tier connections: the WebLogic JDBC driver between the client and WebLogic Server, and the vendor-specific JDBC driver between WebLogic Server and the database.

The advantages to using multitier JDBC are found in the additional services that WebLogic Server provides.

First, some JDBC drivers require vendor-supplied native libraries. The BEA jDriver for Oracle JDBC driver for Oracle databases is one of these. Using the multitier configuration, the native libraries only have to be installed on the computer running WebLogic Server. Clients can use the pure-Java WebLogic JDBC driver instead.

Second, WebLogic Server allows you to create a JDBC connection pool. WebLogic Server creates a set of database connections in the pool. Clients then get a connection from the pool and return it when they are finished. Since the connections are pre-established, the client does not wait for the database connection to be made, a process that can be very time-consuming, especially for e-commerce applications where many clients execute short-lived queries against the database.

Third, a WebLogic Server cluster can provide load-balancing and failover for JDBC connection pools. You assign a DataSource name to a connection pool to register the connection pool in the WebLogic Server replicated naming tree. When clients request a connection from the WebLogic Server cluster, the cluster uses a load-balancing algorithm to select an available connection from one of the servers in the cluster. Even if you are not using a cluster now, you can write your applications to use a DataSource to that you can deploy them later on a cluster with no modifications.

WebLogic Server connection pools have several configurable administrative features. By configuring the maximum number of connections permitted in the pool, you can control the load on the database server. When the demand for connections is heavy, WebLogic Server can increase the number of connections in the pool. When the demand decreases, the additional connections can be dropped, shrinking the pool back to its minimum configuration. Connections can be automatically tested and refreshed to ensure that a client does not get a connection that has gone bad. If the database server goes down, connections are remade automatically when it recovers.

Multitier WebLogic JDBC Applications

WebLogic JDBC provides multitier JDBC services to its clients. The client application gets a JDBC connection from WebLogic Server, and WebLogic Server gets a JDBC connection to the back-end database. Before using WebLogic JDBC, all that is required is that a JDBC driver for the target database is installed on the computer running WebLogic Server and that its Java classes are in the WebLogic Server classpath.

Testing a Multitier JDBC Connection

BEA provides a simple way to test a multitier database connection. Start up WebLogic Server and use the WebLogic **t3dbping** utility to form a multitier connection to a database. **t3dbping** does not use use a connection pool; it just tests whether WebLogic Server can establish a two-tier JDBC connection to a database. It is useful to ensure that the JDBC driver is installed correctly before you set up a connection pool.

If you have a running database server and its JDBC driver installed on the WebLogic Server computer, you can try this with your own database. The example here uses the Cloudscape database. (An evaluation version of the Cloudscape database is included with WebLogic Server so you can try out the WebLogic Server database-dependent examples.)

The following example assumes that you installed WebLogic Server and have not altered the configuration for Cloudscape. If you have problems with Cloudscape, see *Using the Cloudscape Database with WebLogic* at *http://www.weblogic.com/docs51/techsupport/cloudscape.html*.

Here are the steps to test a multitier database connection to Cloudscape with **t3dbping**:

1. Follow the instructions in *Setting your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html* to set up a development shell. This ensures that you will have access to the **t3dbping** class.

2. Start WebLogic Server. See *Setting up and Running WebLogic Server* at *http://www.weblogic.com/docs51/techstart/startserver.html* for help.

3. In your development shell, enter the following command (on one line):

   ```
   java utils.t3dbping t3://localhost:7001 "" "" ""
        COM.cloudscape.core.JDBCDriver
        jdbc:cloudscape:demo
   ```

   If you are using your own database, type `java utils.t3dbping` to see examples for several database vendors.

t3dbping should display the following results:

Listing 6-1 t3dbping output

```
Success!!!

You can connect to the database in your WebLogic JDBC program using:

import java.sql.*;
import java.util.Properties;
import weblogic.common.*;

T3Client   t3   = null;
Connection conn = null;

try {
  t3 = new T3Client("t3://localhost:7001");
  t3.connect();

  Properties dbprops = new Properties();
  dbprops.put("user",     "");
  dbprops.put("password", "");

  dbprops.put("server",   "");
```

```
  Properties t3props = new Properties();
  t3props.put("weblogic.t3.dbprops",        dbprops);
  t3props.put("weblogic.t3",                t3);
  t3props.put("weblogic.t3.driverClassName",
       "COM.cloudscape.core.JDBCDriver");
  t3props.put("weblogic.t3.driverURL",
       "jdbc:cloudscape:demo");

  Class.forName("weblogic.jdbc.t3.Driver").newInstance();
  conn = DriverManager.getConnection("jdbc:weblogic:t3", t3props);

  // Do something with the database connection
  // and then close and disconnect in try blocks

} finally {
  try {conn.close();}    catch (Exception e) {;}
  try {t3.disconnect();} catch (Exception e) {;}
}
```

The output from the **t3dbping** utility includes code that you can use to develop a multitier JDBC Java program. (Note that this example code does not use a connection pool and therefore does not provide the benefits of connection pools, including clustered load-balancing and failover. The use of connection pools is discussed in the next section.) If you do not see these results, **t3dbping** displays a message that should help you diagnose the problem. Another useful debugging utility, **dbping**, lets you test the two-tier JDBC connection on the WebLogic Server computer to diagnose problems in the back-end connection. See *Testing Connections* at *http://www.weblogic.com/docs51/techstart/dbping.html* for help with **dbping**.

The **t3dbping** command has several arguments:

t3:localhost:7001

This argument tells **t3dbping** how to connect to WebLogic Server. It is a URL for a WebLogic Server. *t3* is the protocol to use—the WebLogic T3 protocol. *localhost* is the name of the computer running WebLogic Server. “localhost” is the standard DNS alias for the current computer. If WebLogic Server is running on a different machine on the network, substitute the name of that computer. *7001* is the port where WebLogic Server is listening for standard (unsecure) connections. “7001” is the default port, which can be reconfigured in the `weblogic.properties` file.

"" ""

The first two "" values are for the database *username* and *password.* Cloudscape does not require them, so we marked their place on the **t3dbping** command line with empty strings. Most databases do require a username and password.

""

The third "" value is for the name of the database server or instance, depending on the database vendor’s terminology. With Cloudscape, the database instance is specified with a Java system property, `java.system.property.cloudscape.system.home`, which is defined in the `weblogic.properties` file. So we again mark the position of the argument on the command line with an empty string.

COM.cloudscape.core.JDBCDriver

This is the Java class name of the JDBC driver you want WebLogic Server to load. To find the right class name for your database, you have to consult the JDBC driver documentation.

jdbc:cloudscape:demo

This is a database URL, which the JDBC driver uses to locate the database. The format of the database URL varies for each JDBC driver. Look in the JDBC driver documentation to construct a URL that is correct for your database. With Cloudscape, the third segment of the URL is the database name, in this case “demo.”

Using a WebLogic JDBC Connection Pool

A WebLogic JDBC connection pool opens database connections when WebLogic Server starts up, if the pool is configured in the `weblogic.properties` file, or when the connection pool is created. The connections remain open as long as the connection pool is not destroyed. A client can “reserve” a connection from the pool, which gives the client sole access to the connection until it is returned to the pool. Creating a database connection is an expensive operation on most database systems, and using a connection pool can eliminate most of that expense.

There are three ways to create a connection pool:

- Create a `weblogic.jdbc.connectionPool` property in the `weblogic.properties` file. The connection pool is initialized when WebLogic Server starts.
- Use the WebLogic Console to define the connection pool.
- Create the connection pool in an application, using WebLogic JNDI and the `weblogic.jdbc.common.JdbcServices.createPool()` method. This is called a dynamic connection pool.

Pools created with the WebLogic Console or with Java applications are not automatically recreated when WebLogic Server restarts. Dynamic connection pools can be useful for some applications that need a connection pool only while they are active. An application that creates a dynamic connection pool can also be set up as a WebLogic Server startup class, which has the same effect as defining the connection pool in the `weblogic.properties` file.

Defining a Connection Pool

A sample Cloudscape connection pool is defined (but commented out) in the `weblogic.properties` file included in the WebLogic Server distribution. Here are the properties that define the pool:

```
weblogic.jdbc.connectionPool.demoPool=\
       url=jdbc:cloudscape:demo,\
       driver=COM.cloudscape.core.JDBCDriver,\
       initialCapacity=1,\
       maxCapacity=2,\
       capacityIncrement=1,\
       props=user=none;password=none;server=none
#
# Add an ACL for the connection pool:
weblogic.allow.reserve.weblogic.jdbc.connectionPool.demoPool=\
       everyone
```

The first property defines the connection pool, which is named “demoPool.” The second property specifies which WebLogic users are allowed to reserve a connection from the pool. In this case, specifying the special user “everyone” means that any user can reserve a connection.

The `url` and `driver` arguments are familiar from the **t3dbping** example. The `url`, in the format specified by the JDBC driver vendor, allows the JDBC driver to locate the database. The `driver` is the full class name of the JDBC driver, which can be found in the documentation for the JDBC driver.

The `initialCapacity`, `maxCapacity`, and `capacityIncrement` arguments determine how many database connections the pool contains and when they are created. When the connection pool is first created, WebLogic Server creates `initialCapacity` connections. When all of the connections are in use and another one is requested, WebLogic Server creates `capacityIncrement` new connections and adds them to the pool. The pool will never have more than `maxCapacity` connections.

The `props` argument supplies properties that are passed to the `DriverManager.getConnection()` method when a connection is created.

When you create a connection pool, you should also define a DataSource for the pool, which makes it possible to use the WebLogic JDBC/RMI JDBC driver, which allows a Java client to get a pool connection with a JNDI lookup. To do this, add an additional property, such as this one:

```
weblogic.jdbc.TXDataSource.demoDataSource=demoPool
```

This property causes WebLogic Server to register the connection pool in the JNDI naming tree. The `TXDataSource` portion of the property name tells WebLogic Server to use JTS (Java Transaction Services), which provides transaction support on connections retrieved using this DataSource. If you do not need JTS, you can change this part of the property name to `DataSource`. If your applications use EJB or JMS, however, you should set up a `TXDataSource`.

The `demoPool` property does not use all of the available connection pool configuration options. You can find the entire set of connection pool properties in *Using WebLogic JDBC* at *http://www.weblogic.com/docs51/classdocs/API_jdbct3.html#connpools.*

You can view and change connection pool properties in the WebLogic Console. The Console displays usage statistics for pools, which are useful for tuning connection pool properties.

Using Connection Pools in JDBC Clients

Once a connection pool and DataSource are set up in WebLogic Server, clients get a connection using JNDI.

In this section, we create a simple Java client, `booksPool.java`. This example gets a JNDI InitialContext and looks up the DataSource that provides access to the connection pool. It then creates a `Statement` and executes some SQL commands on it, including some DDL (`create table` and `drop table`) and DML (`insert` and `select`). It shows how to process the results by getting a JDBC `ResultSet` object from the `Statement`. This example also uses a JDBC `PreparedStatement` to perform the SQL `insert` commands. A `PreparedStatement` can be more efficient on the database server for repeated commands because it allows the server to parse and optimize the statement once instead of each time it is executed.

Listing 6-2 booksPool.java

```
package examples.intro;

import java.sql.*;
import java.util.*;
import javax.naming.*;

public class booksPool {

  public static void main(String argv[]) throws Exception {

    Context ctx = null;
    Hashtable ht = new Hashtable();
    java.sql.Connection conn = null;
    ht.put(Context.INITIAL_CONTEXT_FACTORY,
           "weblogic.jndi.WLInitialContextFactory");
    ht.put(Context.PROVIDER_URL,
           "t3://localhost:7001");

    try {
      ctx = new InitialContext(ht);
      javax.sql.DataSource ds =
        (javax.sql.DataSource) ctx.lookup("demoDataSource");
      conn = ds.getConnection();
      Statement s = conn.createStatement();

      try {
        // First, drop the "books" table, in case it
        // already exists. If it does not, then this
        // statement will get an warning that can be ignored.
        System.out.println("Dropping table books.");
        s.execute("drop table books");
      }
      catch ( SQLException se ) {;}


      // Now, create the "books" table and insert some data.
      System.out.println("Creating table books.");
      s.execute("create table books (title varchar(40), " +
                "author varchar(40), publisher varchar(40),"+
                "isbn varchar(20))");

      // Use a PreparedStatement to insert rows in the table.
    System.out.println("Loading table.");
    PreparedStatement ps = conn.prepareStatement(
               "insert into books " +
               "(title,author,publisher,isbn) " +
               " values (?, ?, ?, ?)");

    ps.setString(1, "Java in a Nutshell");
    ps.setString(2, "David Flanagan");
    ps.setString(3, "O’Reilly");
    ps.setString(4, "1-56592-262-X");
    ps.execute();

    ps.setString(1, "Java Examples in a Nutshell");
    //      ps.setString(2, "David Flanagan");
    //      ps.setString(3, "O’Reilly");
    ps.setString(4, "1-56592-371-5");
    ps.execute();

    ps.setString(1, "Design Patterns");
    ps.setString(2, "Gamma, Helm, Johnson, Vlissides");
    ps.setString(3, "Addison-Wesley");
    ps.setString(4, "0-201-63361-2");
    ps.execute();

    System.out.println("Retrieving rows.");
    s.execute("select * from books");

    ResultSet rs = s.getResultSet();
    while (rs.next()) {
      System.out.println(rs.getString("title") + "\n\t" +
                         rs.getString("author") + "\n\t" +
                         rs.getString("publisher") + "\n\t" +
                         rs.getString("isbn"));
    }
    rs.close();
    s.close();
    ps.close();
  } finally {
    try {conn.close();} catch (Exception e) {;}
    try {ctx.close();}  catch (Exception e) {;}
  }
}
```

Running the booksPool Example

To run the `booksPool` example, follow these steps:

1. Set up your development environment as described in *Setting Your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html*.
2. Edit the `weblogic.properties` file and uncomment the `weblogic.jdbc.connectionPool.demoPool` property and the `weblogic.allow.reserve.weblogic.jdbc.connectionPool.demoPool` properties.
3. Define a DataSource for the connection pool by adding this property:

   ```
   weblogic.jdbc.TXDataSource.demoDataSource=demoPool
   ```
4. Start WebLogic Server.
5. In your development shell, change to the directory where you copied the examples.
6. Compile the `booksPool.java` program.

   Windows NT:

   ```
   javac -d %CLIENT_CLASSES% booksPool.java
   ```

   UNIX:

   ```
   javac -d $CLIENT_CLASSES booksPool.java
   ```
7. Run the example with this command:

   ```
   java examples.intro.booksPool
   ```

More about WebLogic JDBC

There are many JDBC features and developer issues that could not be presented in this introduction. Some other JDBC features are illustrated in other sections of this document. For example, the `SqlServlet.java` example in the “Servlets” chapter shows how to use a JDBC connection pool in a server-side class, and how to access database metadata from a ResultSet.

Here are some online documents where you can find out more about JDBC and additional WebLogic JDBC features:

- *WebLogic JDBC Options* at *http://www.weblogic.com/docs51/classdocs/jdbcdrivers.html* summarizes JDBC drivers tested with WebLogic Server, including two-tier and multier JDBC drivers.
- The *JavaSoft JDBC page* at *http://www.javasoft.com/products/jdbc/index.html* has many JDBC documents including APIs, specifications, and white papers.

Documentation for WebLogic JDBC drivers:

- *Using WebLogic jDriver for Oracle* at *http://www.weblogic.com/docs51/classdocs/API_joci.html* describes the BEA Type 2 JDBC driver for Oracle databases.
- *Using WebLogic jDriver for Informix* at *http://www.weblogic.com/docs51/classdocs/API_jinf4.html* describes the BEA Type 4 JDBC driver for Informix Dynamic Server.
- *Using WebLogic jDriver for Microsoft SQL Server* at *http://www.weblogic.com/docs51/classdocs/API_jmsq4.html* describes how to use the BEA Type 4 JDBC driver for Microsoft SQL Server.
- *Using WebLogic JDBC/RMI and Clustered JDBC* at *http://www.weblogic.com/docs51/classdocs/JDBC_RMI.html* describes the JNDI accessible middle-tier JDBC driver that allows Java clients to use connection pools.

CHAPTER

7 EJB (Enterprise JavaBeans)

The Enterprise JavaBeans specification defines a model for component-based applications in Java. WebLogic Server version 5.0 implements the EJB 1.1 specification, including nearly all optional features.

Like JavaBeans, EJBs are reusable software components that can be assembled into applications. But EJBs execute in an EJB Server, WebLogic Server in this case, and they encapsulate business logic instead of GUI objects.

EJB defines separate, independent roles and responsibilities for:

- EJB developers, who design and create Enterprise JavaBeans
- Application developers, who assemble applications using EJBs
- EJB containers, which host EJBs
- System administrators, who deploy EJBs in an EJB server

An *EJB developer* creates an EJB by defining a public EJB interface for the bean and implementing the interfaces for the desired EJB characteristics.

An *application developer* uses EJBs by calling standard JNDI and EJB methods and the EJB's public interface methods.

An *EJB container*, such as WebLogic Server, manages an EJB through its life cycle and provides run-time services, such as caching, transaction management, persistence, and tools that support EJB deployment.

System administrators deploy EJBs in the servers they manage. They configure runtime properties for each EJB by modifying a deployment descriptor. The deployment descriptor supplies the EJB server with the information it requires to host the EJB. The deployment descriptor also allows the system administrator to define permissions on an EJB and to configure resources used by an EJB.

The EJB specification establishes "contracts" that define the services provided by each role. Be sure to *read the EJB specification*, which you can download from Sun's EJB documentation page at *http://java.sun.com/products/ejb/docs.html*, before you begin developing or using EJBs. By defining the roles and responsibilities for creating, deploying, hosting, and accessing EJBs, the EJB component model enables you to create your own EBJs or obtain EJBs from other sources and easily incorporate them into applications.

EJBs depend upon, and utilize, other J2EE technologies. For example, you use JNDI to access an instance of an EJB in applications, such as HTTP Servlets or JSP pages. EJBs can use JDBC for persistence and they can use JTA to participate in transactions involving other transactional J2EE technologies such as JMS. They can use RMI or IIOP (CORBA) to operate in a distributed environment.

Types of EJBs

There are two types of EJBs: session beans and entity beans. Each of these has subtypes, which are described in the following sections.

Session Beans

A session bean is a transient EJB instance that serves a single client. The EJB container creates a session bean at a client's request and maintains the bean as long as the client maintains its connection to the bean. Sessions beans are not persistent. Session beans tend to implement procedural logic; they embody actions more than data.

A session bean can be stateless or stateful. Stateless session beans maintain no client-specific state between calls and can be used by any client. They can be used to provide access to services that do not depend on the context of a session, such as sending a document to a printer or retrieving non-updateable data into an application.

A stateful session bean maintains some state on behalf of a specific client. Stateful session beans can be used to manage a process, such as assembling an order or routing a document through a workflow process. With the ability to accumulate and maintain state through multiple interactions with a client, session beans are often the controlling objects in an application. Since they are not persistent, session beans must complete their work in a single session and use JDBC, JMS, or entity beans to record the work permanently.

Entity Beans

An entity bean represents a dataful object, such as a customer, an account, or an inventory item. Entity beans contain data values and methods to act upon those values. The values are saved in a database (using JDBC) or in a file (as a serializable Java object), or in some other data store. Entity beans can participate in transactions involving other EJBs and transactional services, such as JMS.

Entity beans are often mapped to objects in databases. An entity bean could represent a row in a table, a single column in a row, or an entire table or query result. Application requirements determine how beans are mapped to database objects.

An entity bean can employ bean-managed persistence, where the bean contains code to retrieve and save persistent values, or container-managed persistence, where the EJB container loads and saves values on behalf of the bean. With container-managed persistence, the WebLogic EJB compiler can generate JDBC support classes to map an entity bean to a row in a database. Other container-managed persistence mechanisms are available. For example, TOPLink for BEA WebLogic Server, from *The Object People* (*http://www.objectpeople.com*), provides persistence for an object relational database.

Entity beans may be shared by many clients and applications. An instance of an entity bean can be created at the request of some client, but it does not disappear when that client disconnects. It continues to live as long as any client is actively using it. When the bean is no longer in use, the EJB container may passivate it—remove the live instance from the server. To minimize accesses to the persistence store, and to optimize server performance, the EJB server administrator can tune the frequency of passivation in a bean's deployment descriptor. For example, the system administrator can specify the maximum number of bean instances to cache and how long to delay passivation when the bean is no longer in use. By configuring each bean independently, the system administrator can tune a WebLogic Server for the combination of applications that share it.

Developing EJBs

Developing an EJB involves implementing interfaces from the `javax.ejb` package and defining and implementing the bean's public interface. The EJB specification tells you what classes, interfaces, and methods are required for each type of bean. The specification also contains rules that must be followed to ensure that the bean can be deployed in any compliant EJB server.

Some EJB code is generated with tools provided with the EJB server. The WebLogic EJB compiler, `weblogic.ejbc`, generates this code for WebLogic Server. WebLogic Server also includes DeployerTool, a graphical utility that makes it easy to change WebLogic Server deployment properties, validate beans against the EJB 1.1 specification, and deploy beans in WebLogic Servers.

The `Emp` example shows how to build a JDBC container-managed entity bean for an existing database table. This example uses the `Emp` table, which is in the Cloudscape `demo` database included in the WebLogic Server distribution.

The `Emp` table has the following SQL definition:

```
create table emp (
       empno     int not null,
       ename     varchar(10),
       job       varchar(9),
       mgr       int,
       hiredate  date,
       sal       float,
       comm      float,
       deptno    int
       )
```

The `Emp` Enterprise JavaBean example includes these source files:

`EmpBeanHome.java extends javax.ejb.EJBHome`
: Defines the home interface for the `Emp` bean, including one `create()` signature, and four finder methods.

`Emp.java extends javax.ejb.EJBObject`
: The public interface for the `Emp` EJB. It defines a get method for each field, a set method for each field except the primary key (`empno`), and an `htmlToPrint()` method that returns a String representing an `Emp` EJB instance in HTML.

`EmpBean.java implements javax.ejb.EntityBean`
: Contains the implementation for the `Emp` EJB and the `EntityBean` methods required by the EJB specification. It declares public variables corresponding to the Emp database table columns. It implements the required `ejbCreate()` method and overrides several life cycle methods, which are defined in the `javax.ejb.EntityBean` interface. Finally, it implements each method defined by the `Emp` interface.

`EmpBeanPK.java implements java.io.Serializable`
: The primary key class for the `Emp` EJB. The primary key for this database table is the `empno` column. This class defines the default constructor and a constructor that sets the primary key field for an instance of this class.

`ejb-jar.xml`
`weblogic-ejb-jar.xml`
`weblogic-cmp-rdbms-jar.xml`

These are XML deployment files. `ejb-jar.xml` contains deployment parameters defined by the EJB specification. `ejb-jar.xml` references `weblogic-ejb.jar.xml` and `weblogic-cmp-rdbms-jar.xml`. `weblogic-ejb-jar.xml` contains EJB deployment information that is specific to to the WebLogic Server EJB container. `weblogic-cmp-rdbms-jar.xml` contains WebLogic Server-specific deployment information to map an entity bean to a database using WebLogic Server's container-managed persistence.

Remote Interface

The remote interface defines a public interface for an EJB. The remote interface for the `Emp` bean is defined in the `Emp.java` file:

Listing 7-1 Emp.java

```
package examples.intro;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import java.sql.Date;

public interface Emp extends EJBObject {

  public int getEmpno()
      throws EJBException, RemoteException;
  public String getEname()
      throws EJBException, RemoteException;
  public String getJob()
      throws EJBException, RemoteException;
  public int getMgr()
      throws EJBException, RemoteException;
  public java.sql.Date getHiredate()
      throws EJBException, RemoteException;
  public double getSal()
      throws EJBException, RemoteException;
  public double getComm()
      throws EJBException, RemoteException;
  public int getDeptno()
      throws EJBException, RemoteException;

  public void setEname(String newEname)
      throws EJBException, RemoteException;
  public void setJob(String newJob)
      throws EJBException, RemoteException;
  public void setMgr(int newMgr)
      throws EJBException, RemoteException;
  public void setHiredate(Date newHiredate)
       throws EJBException, RemoteException;
  public void setSal(double newSal)
      throws EJBException, RemoteException;
  public void setComm(double newComm)
      throws EJBException, RemoteException;
  public void setDeptno(int newDeptno)
      throws EJBException, RemoteException;

  public String htmlToPrint()
      throws EJBException, RemoteException;
}
```

The interface defines one `get` method and one `set` method for each field. The data types correspond to the JDBC data types of the database table. The `htmlToPrint()` method returns a `String` with the employee's data coded as a row in an HTML table. The string can be directed to the output of an HTTP Servlet or JSP page.

Implementation Class

The implementation for the Emp bean is in the `EmpBean.java` file. The bean implements the `javax.ejb.EntityBean` interface, which defines several methods that an entity bean must implement. These methods are called by the EJB container (WebLogic Server), at various times during the bean's life cycle. The `EmpBean` class also implements the methods defined by the `Emp` interface.

Listing 7-2 EmpBean.java

```
package examples.intro;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;
import java.sql.Date;

public class EmpBean implements EntityBean {

  protected EntityContext entityContext;

  public int empno;
  public String ename;
  public String job;
  public int mgr;
  public Date hiredate;
  public double sal;
  public double comm;
  public int deptno;

  public EmpBeanPK ejbCreate(int v_empno, String v_ename,
                      String v_job, int v_mgr,
                      Date v_hiredate, double v_sal,
                      double v_comm, int v_deptno)
    throws EJBException, CreateException {
    empno = v_empno;
    ename = v_ename;
    job = v_job;
    mgr = v_mgr;
    hiredate = v_hiredate;
    sal = v_sal;
    comm = v_comm;
    deptno = v_deptno;
    return new EmpBeanPK(empno);
  }

  public void ejbPostCreate(int v_empno, String v_ename,
                          String v_job, int v_mgr,
                          Date v_hiredate, double v_sal,
                          double v_comm, int v_deptno)
    throws EJBException
  {
    // do nothing
  }

  public void ejbActivate() throws EJBException {
    // do nothing
  }

  public void ejbLoad() throws EJBException {
    // do nothing
  }
```

```
public void ejbPassivate() throws EJBException {
  // do nothing
}

public void ejbRemove() throws EJBException {
  // do nothing
}

public void ejbStore() throws EJBException {
  // do nothing
}

public void setEntityContext(EntityContext ctx)
    throws EJBException {
  entityContext = ctx;
}

public void unsetEntityContext() {
  entityContext=null;
}

public int getEmpno()
    throws EJBException, RemoteException {
  return empno;
}

public String getEname()
    throws EJBException, RemoteException {
  return ename;
}

public String getJob()
    throws EJBException, RemoteException {
  return job;
}

public int getMgr()
    throws EJBException,RemoteException {
  return mgr;
}

public Date getHiredate()
    throws EJBException, RemoteException {
  return hiredate;
}

public double getSal()
    throws EJBException, RemoteException {
  return sal;
```

```
}

public double getComm()
    throws EJBException, RemoteException {
  return comm;
}

public int getDeptno()
    throws EJBException, RemoteException {
  return deptno;
}

public void setEname(String newEname)
    throws EJBException, RemoteException {
  ename = newEname;
}

public void setJob(String newJob)
    throws EJBException, RemoteException {
  job = newJob;
}

public void setMgr(int newMgr)
    throws EJBException, RemoteException {
  mgr = newMgr;
}

public void setHiredate(Date newHiredate)
    throws EJBException, RemoteException {
  hiredate = newHiredate;
}

public void setSal(double newSal)
    throws EJBException, RemoteException {
  sal = newSal;
}

public void setComm(double newComm)
    throws EJBException, RemoteException {
  comm = newComm;
}

public void setDeptno(int newDeptno)
    throws EJBException, RemoteException {
  deptno = newDeptno;
}

public String htmlToPrint()
    throws EJBException, RemoteException {
```

```
    String work = "<tr>" +
      "<td>" + getEmpno()     + "</td>" +
      "<td>" + getEname()    + "</td>" +
      "<td>" + getJob()      + "</td>" +
      "<td>" + getMgr()      + "</td>" +
      "<td>" + getHiredate() + "</td>" +
      "<td>" + getSal()      + "</td>" +
      "<td>" + getComm()     + "</td>" +
      "<td>" + getDeptno()   + "</td></tr>";
    return work;

  }
}
```

`EmpBean.java` implements methods from the `javax.ejb.EntityBean` interface. WebLogic Server calls these methods when an EJB moves from one state to another, for example when the bean moves from active to passive state or to and from the database. These methods do not have any specified behaviors; they are provided so you can perform any actions required when an event changes the bean's status.

In a bean-managed persistence entity bean, the `ejbLoad()` and `ejbStore()` methods contain the code to load or save a bean. In `Emp`, these are notification callbacks that are called after loading or before storing the bean to the database.

An `ejbCreate()` method is called when a bean is created. You must supply an `ejbCreate()` method to match each `create()` method you define in the bean's home interface. This example has one `ejbCreate()` method that sets all of the fields in the bean. A matching `postCreate()` method is called after the bean is created to allow the bean to complete any initialization activity required.

The get methods simply return the value of a field. The set methods change the field value.

Home Interface

The home interface defines create methods, which behave like constructors for the bean, and finder methods, which the container calls with application-supplied criteria to retrieve a bean instance or an enumeration of bean instances from the persistence store. The home interface for the `Emp` bean is in the `EmpBeanHome.java` file:

Listing 7-3 EmpBeanHome.java

```
package examples.intro;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;
import java.sql.Date;

public interface EmpBeanHome extends EJBHome {

    public Emp create(int empno, String ename, String job,
                      int mgr, Date hiredate, double sal,
                      double comm, int deptno)
       throws RemoteException, EJBException, CreateException;

    public Emp findByPrimaryKey(EmpBeanPK pk)
       throws RemoteException, FinderException;

    public Emp findByEmpno(int empno)
       throws RemoteException, FinderException;

    public Enumeration findByName(String name)
       throws RemoteException, FinderException;

    public Enumeration findByNameLike(String name)
       throws RemoteException, FinderException;

}
```

This interface defines one `create()` method and four finder methods. The `findByPrimaryKey()` finder method takes an instance of the bean's primary key class as its argument. The `findByEmpNo()` method is equivalent to the `findByPrimaryKey()` method. It returns a single `Emp` instance because the `empno` field is unique. The `findByName()` and `findByNameLike()` finders return Enumerations, since either method could match more than one employee.

In `Emp`, the JDBC code that implements these methods is generated by the WebLogic EJB compiler using this interface and definitions in the bean's deployment descriptor.

Primary Key Class

Entity beans must have a unique primary key to distinguish bean instances. The primary key is constructed from one or more fields and is described in a class that implements `java.io.Serializable`. The fields that represent the primary key must be declared public members of the primary key class and the class must have a public default constructor (with no arguments). The home interface includes a finder method, `findByPrimaryKey()`, which uses an instance of this class to retrieve a specific bean.

The primary key class must define `hashCode()` and `equals()` methods. The `hashCode()` method in this example just returns the employee number, which, conveniently, is a unique integer for each employee.

The primary key class for the `Emp` bean is in the `EmpBeanPK.java` file:

Listing 7-4 EmpBeanPK.java

```
package examples.intro;

import javax.ejb.*;
import java.util.*;
import java.lang.Integer;

public class EmpBeanPK implements java.io.Serializable {

  public int empno;
  public EmpBeanPK(int empno) {this.empno = empno; }

  public EmpBeanPK() { }
  public String toString() {
    Integer n = new Integer(empno);
    return n.toString();
  }

  public int hashCode() {
    return empno;
  }

  public boolean equals(Object other) {
    return ((EmpBeanPK) other).empno == this.empno;
  }
}
```

Deployment Descriptors

Deployment descriptors identify the Java classes that implement the bean, the name used to bind the bean in the WebLogic Server JNDI tree, transaction requirements, security properties, caching and passivation attributes and, for container managed-entity beans, attributes to map the bean to database objects and finder methods.

Deployment descriptors are defined using XML and packaged with the bean in an EJB `.jar` file. The easiest way to edit deployment information for an EJB is to use the DeployerTool. See *Deploying EJBs in WebLogic Server* at *http://www.weblogic.com/docs/50/classdocs/API_ejb/EJB_deployover.html* for help with the XML deployment descriptors.

The EJB 1.1 specification defines the contents of the `ejb-jar.xml` file. Here is the `ejb-jar.xml` file for the Emp bean:

Listing 7-5 ejb-jar.xml

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_2.dtd'>

<ejb-jar>
    <enterprise-beans>
      <entity>
       <ejb-name>Emp</ejb-name>
       <home>examples.intro.EmpBeanHome</home>
       <remote>examples.intro.Emp</remote>
       <ejb-class>examples.intro.EmpBean</ejb-class>
       <persistence-type>Container</persistence-type>
       <prim-key-class>examples.intro.EmpBeanPK</prim-key-class>
       <reentrant>False</reentrant>
       <cmp-field>
         <field-name>hiredate</field-name>
       </cmp-field>
       <cmp-field>
         <field-name>ename</field-name>
       </cmp-field>
       <cmp-field>
         <field-name>sal</field-name>
       </cmp-field>
       <cmp-field>
```

```
          <field-name>comm</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>mgr</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>empno</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>deptno</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>job</field-name>
        </cmp-field>
       </entity>
     </enterprise-beans>
     <assembly-descriptor>
       <container-transaction>
        <method>
          <ejb-name>Emp</ejb-name>
          <method-intf>Remote</method-intf>
          <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
       </container-transaction>
     </assembly-descriptor>
   </ejb-jar>
```

The Emp bean uses two other XML deployment files: `weblogic-ejb-jar.xml`, which contains WebLogic Server-specific deployment parameters, and `weblogic-cmp-rdbms-xml.jar`, which contains parameters that configure a container-managed persistence bean for a database. The `weblogic-ejb-jar.xml` file for the Emp bean provides caching parameters, and, in the `<persistence-descriptor>` section, specifies that the bean uses WebLogic container-managed persistence. Here is `weblogic-ejb-jar.xml`:

Listing 7-6 weblogic-ejb-jar.xml

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC ’-//BEA Systems, Inc.//DTD
WebLogic 5.1.0 EJB//EN’
’http://www.beasys.com/j2ee/dtds/weblogic-ejb-jar.dtd’>
```

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
      <ejb-name>Emp</ejb-name>
      <caching-descriptor>
       <max-beans-in-free-pool>20</max-beans-in-free-pool>
       <initial-beans-in-free-pool>0</initial-beans-in-free-pool>
       <max-beans-in-cache>100</max-beans-in-cache>
       <idle-timeout-seconds>10</idle-timeout-seconds>
       <cache-strategy>Read-Write</cache-strategy>
      </caching-descriptor>
      <persistence-descriptor>
       <persistence-type>
         <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
         <type-version>5.1.0</type-version>
<type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-storage>
       </persistence-type>
       <persistence-use>
         <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
         <type-version>5.1.0</type-version>
       </persistence-use>
      </persistence-descriptor>
      <jndi-name>EmpBeanHome</jndi-name>
    </weblogic-enterprise-bean>
  </weblogic-ejb-jar>
```

The `weblogic-cmp-rdbms-jar.xml` file maps an entity bean into a database. It names the JDBC connection pool, the table in the database, and then maps each EJB field to a database column. The `<finder-list>` section defines a query for each of the bean's finder methods, using WebLogic Query Language (WQL). Here is `weblogic-cmp-rdbms-jar.xml`:

Listing 7-7 weblogic-cmp-rdbms-jar.xml

```
<!DOCTYPE weblogic-rdbms-bean PUBLIC
 "-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS
Persistence//EN"
 "http://www.beasys/com/weblogic-rdbms-persistence.dtd">
<weblogic-rdbms-bean>
  <pool-name>demoPool</pool-name>
  <table-name>emp</table-name>
  <attribute-map>
    <object-link>
      <bean-field>mgr</bean-field>
      <dbms-column>mgr</dbms-column>
      </object-link>
      <object-link>
        <bean-field>hiredate</bean-field>
        <dbms-column>hiredate</dbms-column>
      </object-link>
      <object-link>
        <bean-field>job</bean-field>
        <dbms-column>job</dbms-column>
      </object-link>
      <object-link>
        <bean-field>comm</bean-field>
        <dbms-column>comm</dbms-column>
      </object-link>
      <object-link>
        <bean-field>sal</bean-field>
        <dbms-column>sal</dbms-column>
      </object-link>
      <object-link>
        <bean-field>deptno</bean-field>
        <dbms-column>deptno</dbms-column>
      </object-link>
      <object-link>
        <bean-field>empno</bean-field>
        <dbms-column>empno</dbms-column>
      </object-link>
      <object-link>
        <bean-field>ename</bean-field>
        <dbms-column>ename</dbms-column>
      </object-link>
    </attribute-map>
    <finder-list>
      <finder>
        <method-name>findByNameLike</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
        <finder-query><![CDATA[(like ename $0)]]></finder-query>
      </finder>
      <finder>
        <method-name>findByEmpno</method-name>
        <method-params>
          <method-param>int</method-param>
        </method-params>
        <finder-query><![CDATA[(= empno $0)]]></finder-query>
      </finder>
      <finder>
        <method-name>findByName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
    </method-params>
    <finder-query><![CDATA[(= ename $0)]]></finder-query>
  </finder>
</finder-list>
<options>
  <use-quoted-names>false</use-quoted-names>
</options>
</weblogic-rdbms-bean>
```

Running the Emp Example

1. Set up your development environment as described in *Setting Your Development Environment* at *http://www.weblogic.com/docs51/techstart/environment.html*.
2. Change to the `examples/intro` directory, and create a temporary directory named `ejbtemp` and then a directory named `ejbtemp/META-INF:`

   ```
   mkdir ejbtemp
   ```

   ```
   mkdir ejbtemp/META-INF
   ```
3. Copy the deployment descriptor files to the `ejbtemp/META-INF` directory:

   ```
   copy *.xml ejbtemp/META-INF
   ```
4. Compile the Java source files. Enter this command on a single line:

   ```
   javac -d ejbtemp Emp.java EmpBean.java EmpBeanPK.java
       EmpBeanHome.java
   ```

   This command compiles the Emp bean into the `ejbtemp` directory, which is used to “stage” the EJB.

   An enterprise bean is packaged in a jar file. To create the jar file, you create a directory structure with all of the files you want to include. Then when you run the **jar** program, the directory structure is duplicated in the jar file.
5. Compile the Java source files again, this time into the directory specified by the `CLIENT_CLASSES` environment variable.

   Windows NT:

   ```
   javac -d %CLIENT_CLASSES% Emp.java EmpBean.java EmpBeanPK.java
       EmpBeanHome.java
   ```

UNIX:

```
javac -d $CLIENT_CLASSES Emp.java EmpBean.java EmpBeanPK.java
    EmpBeanHome.java
```

This **javac** command compiles the classes into a directory that is in the classpath, which allows the WebLogic EJB compiler to find them later.

6. Change to the `ejbtemp` directory and create a `jar` file named `Emp.jar`:

   ```
   cd ejbtemp
   jar cf Emp.jar examples META-INF
   ```

7. Run the WebLogic EJB compiler on the `Emp.jar` file you created:

   Windows NT:

   ```
   java -Dweblogic.home=%WL_HOME% weblogic.ejbc Emp.jar
           -d %WL_HOME%\myserver\Emp.jar
   ```

   UNIX:

   ```
   java -Dweblogic.home=$WL_HOME weblogic.ejbc Emp.jar
           -d $WL_HOME/myserver/Emp.jar
   ```

   The EJB compiler creates a new `Emp.jar` file in the `./myserver` directory of the WebLogic Server. This jar file contains the classes you compiled, plus classes generated by the EJB compiler.

8. Edit the `weblogic.properties` file and find the `weblogic.ejb.deploy` property. Add the full path for the `Emp.jar` file to this property. For example, if you installed WebLogic Server in the `c:\weblogic` directory, this property would be:

   ```
   weblogic.ejb.deploy=\
        c:/weblogic/myserver/Emp.jar
   ```

9. Start (or restart) WebLogic Server.

10. You can delete the `examples/intro/ejbtemp` directory and all of its contents.

You should see a message in the WebLogic Server log to tell you that the bean has been deployed.

The next section shows how to use the Emp bean in an application.

Using the Bean in an Application

The `EmpQuery.jsp` JSP page is a browser-based lookup form that lets you query the Emp table using the Emp bean's finder methods. It demonstrates how to look up a bean with JNDI, how to call the finder methods, and how to use the bean's `htmlToPrint()` method to include the bean's fields in an HTML table.

Here is a listing of the `EmpQuery.jsp` file:

Listing 7-8 EmpQuery.jsp

```
<!doctype html p

ublic "-//w3c/dtd HTML 4.0//en">
<html>
<head><title>Employee lookup</title></head>

<%@ page
  info="EmpBean lookup example"
  contentType="text/html"
  import="java.lang.Integer, javax.naming.*, javax.ejb.*,
weblogic.common.*,
          examples.intro.Emp, examples.intro.EmpBeanHome"
%>
<body>
<font face="Helvetica">
<h1><font color="#FF0000">Employee Lookup</font></h1>

<hr>
<form action=<%= request.getRequestURI() %> method="post">
<table border=0 cellspacing=5 align=center>
<tr>
  <td>Search by</td>
  <td><select name=searchType>
  <option value=1>Employee number</option>
  <option selected value=2>Employee name</option>
  <option value=3>Name like ('%' is wildcard)</option>
  </select>
  </td>
</tr>
<tr>
  <td>Search for</td>
  <td><input type=text name=criteria size=10 maxlength=10></td>
</tr>
```

```
<tr>
  <td align=center colspan=2><input type=submit value="Go">
  </td>
</tr>
</table>
</form>
<p>

<%

    if (request.getParameter("searchType") != null
       && request.getParameter("criteria") != null)
    {
      int searchType =
Integer.parseInt(request.getParameter("searchType"));
      String criteria = request.getParameter("criteria");
      Emp emp = null;
      Enumeration emps = null;
      try {
        Context ctx = new InitialContext();
       EmpBeanHome home = (EmpBeanHome) ctx.lookup("EmpBeanHome");
        switch (searchType) {
        case 1: // by employee number
         emp = (Emp) home.findByEmpno(Integer.parseInt(criteria));
          break;
        case 2: // by Employee name
          emps = home.findByName(criteria.toUpperCase());
          break;
        case 3: // by name like
          emps = home.findByNameLike(criteria.toUpperCase());
          break;
        default:
        }

        if (emp != null || emps != null) {
          // we have data
%>

<hr>
<h2>Search Results</h2>
<table border=1 align=center>
<tr>
  <th>Employee number</th>
  <th>Name</th>
  <th>Title</th>
  <th>Manager</th>
  <th>Hire date</th>
  <th>Salary</th>
  <th>Commission</th>
  <th>Department</th>
</tr>

<%
       }
       if (emp != null) {
         out.print(emp.htmlToPrint());
       }
       else {
         while (emps.hasMoreElements()) {
           emp = (Emp) emps.nextElement();
           out.print(emp.htmlToPrint());
         }
       }
%>
</table>

<%
   }
      catch (NumberFormatException nfe) {
        out.print("<p>Error attempting to convert " +
                  criteria + " to an integer.");
      }
      catch (Exception e) {
        out.print("<p>Exception: " + e.getMessage());
      }
    }
%>
<p>
<hr>
<center>
<p>Executed by
<%= application.getServerInfo() %>.<br>
Copyright 1999-2000 (C) BEA Systems, Inc.
All Rights Reserved.
</center>
</font>
</body>
</html>
```

You use JNDI to look up the bean's home interface:

```
    Context ctx = new InitialContext();
    EmpBeanHome home = (EmpBeanHome) ctx.lookup("EmpBeanHome");
```

Then, using the home interface, you can retrieve instances by using the bean's finder methods, or you could create a new instance using the `create()` method of the home interface. This example uses the `findByEmpno()`, `findByName()`, or `findByNameLike()` methods to retrieve beans, depending on user's browser entries.

The `findByEmpno()` method returns a single instance of an Emp bean while the `findByName()` and `findByNameLike()` methods return Enumerations of Emp beans. In either case, you must cast the returned object to the `Emp` interface, and then you can use any of the methods defined by that interface.

Running the EmpQuery.jsp Example

To try the `EmpQuery.jsp` example, make sure you have deployed the Emp bean as described in the previous section and that WebLogic Server is running.

1. Copy `EmpQuery.jsp` to the `myserver/public_html` directory of your WebLogic Server installation.
2. In a browser, enter the URL for `EmpQuery.jsp`. For example:

   ```
   http://localhost:7001/EmpQuery.jsp
   ```

   Change the hostname and port number if you are running WebLogic Server on a different computer or at a different listen port.
3. Choose the type of lookup, enter the value to search for, and click **Go**.

The "Name Like" search type employs the SQL `LIKE` clause, which uses the percent sign (%) as a wildcard. For example, to find all employees with names starting with 'A', enter `"A%"` in the "Search for" field. You can list *all* employees by entering `"%"` in the "Search for" field with a "Name like" search type.

More about EJB

The Sun *Enterprise JavaBeans 1.1 specification* at *http://java.sun.com/products/ejb/docs.html* is required reading for EJB developers. You can find javadocs for the `javax.ejb` interfaces and classes on the same web page.

Read the WebLogic EJB developers guide, *Using WebLogic EJB* at *http://www.weblogic.com/docs51/classdocs/API_ejb/index.html* for more about creating and deploying EJBs in WebLogic Server.

CHAPTER

8 JMS (Java Message Service)

Java Message Service (JMS) allows Java programs to exchange messages with other Java programs sharing a messaging system. A messaging system accepts messages from "producer" clients and delivers them to "consumer" clients.

Messaging systems, sometimes called Message-Oriented Middleware (MOM), enable Java clients to use their services by supplying a Java layer called a JMS Provider, which implements JMS for the specific product.

WebLogic JMS implements the *JMS specification version 1.0.1*, which is online at *http://www.javasoft.com/products/jms/docs.html*. WebLogic JMS includes a full-featured messaging system, which can be configured by setting properties in the `weblogic.properties` file, from the WebLogic Console, or programmatically, using the JMS interfaces.

You can use WebLogic JMS with the other WebLogic Server APIs and facilities, such as Enterprise Java Beans, JDBC connection pools, HTTP Servlets, JSP pages and RMI. JMS operations can participate in transactions with other Java APIs that use the Java Transaction API.

JMS Messaging Models

JMS supports two messaging models: point-to-point (PTP) and publish/subscribe. The terms "message producer" and "message consumer" describe clients that send and receive messages in either of the models, although each model has its own specific terms for producers and consumers.

Point-to-Point Messaging

The point-to-point messaging model is based on message Queues. A QueueSender (producer) sends a message to a specified Queue. A QueueReceiver (consumer) receives messages from the Queue. A Queue can have multiple QueueSenders and QueueReceivers, but an individual message can only be delivered to one QueueReceiver. If multiple QueueReceivers are listening for messages on a Queue, WebLogic JMS determines which will receive the next message. If no QueueReceivers are listening on the Queue, messages remain in the Queue until a QueueReceiver attaches to the Queue.

Publish/Subscribe Messaging

The publish/subscribe messaging model is organized around Topics. TopicPublishers (producers) send messages to a Topic. TopicSubscribers (consumers) retrieve messages from a Topic. Unlike the point-to-point model, many TopicSubscribers can receive the same message.

A durable subscriber is a feature of publish/subscribe messaging. A durable subscriber allows you to create a named TopicSubscriber, usually associated with a user or application. JMS retains messages until all TopicSubscribers have received them.

JMS Persistence

Messages can be persistent or non-persistent. WebLogic JMS writes persistent messages to a database via a JDBC connection pool you assign to JMS in the `weblogic.properties` file. A persistent message is not considered sent until it has been stored in the database. Persistent messages are guaranteed to be delivered *at least* once. Non-persistent messages are not stored in a database and so they may be lost during a failure. A non-persistent message is guaranteed to be delivered *at most* once.

JMS Classes and Interfaces

JMS defines several objects that allow you to send and receive messages. JMS objects are subclassed from common parent classes to provide Queue- and Topic-specific versions of the classes.

This section describes the most important JMS classes. See the *javax.jms javadocs* at *http://www.javasoft.com/products/jms/javadoc-101a/index.html* for complete descriptions of all JMS classes.

ConnectionFactory

You access JMS initially using a ConnectionFactory, which is bound in the WebLogic Server JNDI tree. You look up a QueueConnectionFactory or a TopicConnectionFactory and then use it to create a Connection.

Connection

A Connection (QueueConnection or TopicConnection) manages all of the messaging activity between a JMS client and a JMS provider. It is also a factory for Session objects. A new Connection is stopped—no messages flow until you start the Connection by calling its `start()` method.

Session

A Session (QueueSession or TopicSession) is a context for producing and consuming messages. Sessions create message consumers and producers and manage the flow of messages, including the ability to group send and receive operations into transactions.

Message Producer

A message producer (QueueSender or TopicProducer) transmits messages from a JMS client to a destination (Queue or Topic). A message producer is associated with the destination Queue or Topic when it is created.

Destination

A destination (Queue or Topic) is the object that receives and distributes messages. Destinations are bound in the JNDI tree and can be defined in the `weblogic.properties` file or in the WebLogic Console. An application can create a TemporaryQueue or TemporaryTopic that exists only as long as the Connection that creates it.

Message Consumer

A **message consumer** (QueueReceiver or TopicSubscriber) receives messages from a destination. A consumer is created by calling the `CreateReceiver()` or `CreateSubscriber()` method of the Session. Messages can be received asynchronously by providing a class that implements the MessageListener interface. Messages are forwarded to the `onMessage()` method of the MessageListener. Messages can be received synchronously by calling a `receive()` method on the consumer. There are `receive()` methods that return immediately if no message is waiting, wait for a specified period of time, or wait indefinitely for a message.

Message

JMS messages have three parts. The message header contains fields that the JMS system uses to describe and deliver messages. Message headers are also available to applications. The message properties section contains application-defined properties that can be attached to a message. These properties, and the message headers, can be referenced in selectors, which allow a consumer to filter for messages using criteria resembling an SQL `where` clause. The message body holds the contents of the message.

JMS supports five types of message bodies. The simplest of these is a TextMessage which holds a single Java String value. A BytesMessage holds a stream of uninterpreted bytes and a StreamMessage holds a stream of Java primitive types. BytesMessage and StreamMessage are written and read using methods similar to those of the `java.io.DataOutputStream` and `java.io.DataInputStream` classes. A MapMessage holds name/value pairs similar to a Hashtable. The values can be read randomly by specifying the name, or they can be returned in an Enumeration. An ObjectMessage holds any serializable Java object.

Using JMS

Some JMS objects are "administered," which means they must be configured in WebLogic Server before they can be used by applications. The administered objects are QueueConnectionFactories, TopicConnectionFactories, Queues, and Topics. If you plan to use persistent messages, you must also specify a JDBC connection pool where JMS will store data. If you want to use Enterprise JavaBeans and JMS persistence in the same transaction, the two must use the same connection pool.

JMS administered objects can be configured using the WebLogic Console, but to make them permanent, you define them in the `weblogic.properties` file. See *Configuring WebLogic JMS* at *http://www.weblogic.com/docs51/classdocs/API_jms.html#configuring* for instructions for configuring JMS in your WebLogic Server.

Sending Messages

To send messages to a JMS Queue or Topic, you use JNDI to look up a ConnectionFactory and the destination Queue or Topic. Using the ConnectionFactory, you create a series of JMS objects, including a Connection, Session, producer (QueueSender or TopicPublisher), and a Message. Then you start the Connection and begin sending messages. Although JMS requires several objects, the send process is very straightforward.

The `EmpTrans.java` example is an HTTP Servlet that demonstrates how to send messages to a JMS Queue. This example is one part of a rudimentary workflow system. The servlet displays an HTML form and collects data for a transaction against the Emp table. It creates a JMS MapMessage containing the data entered and sends the message to the EmpXactQueue Queue, using the `persistent` delivery mode so that the message is saved in the JMS database.

The JMS code is in the `sendRequest()` method of the `EmpTrans.java` example.

Listing 8-1 EmpTrans.sendRequest()

```
private boolean sendRequest(
       int xactType, int empno,
       String ename, String job,
       int mgr, java.sql.Date hiredate,
       float sal, float comm, int deptno)
  {

    boolean result = true;

    Context ctx = null;
    QueueConnectionFactory factory;
    QueueConnection qconnection = null;
    QueueSession qsession = null;
    QueueSender qsender = null;
    Queue queue;
    MapMessage message;
    Hashtable ht = new Hashtable();

    ht.put(Context.SECURITY_PRINCIPAL, "guest");
    ht.put(Context.SECURITY_CREDENTIALS, "guest");
    try {
      ctx = new InitialContext(ht);
      factory = (QueueConnectionFactory)
        ctx.lookup("javax.jms.QueueConnectionFactory");
      qconnection = factory.createQueueConnection();
      qsession = qconnection.createQueueSession(false,
                        Session.AUTO_ACKNOWLEDGE);

      queue = (Queue) ctx.lookup("jms.queue.EmpXactQueue");
      qsender = qsession.createSender(queue);
      message = qsession.createMapMessage();
      qconnection.start();

      message.setInt("xactType", xactType);
      switch (xactType) {
      case 1: // hire
        message.setString("ename", ename);
        message.setString("job", job);
        message.setInt("mgr", mgr);
        message.setString("hiredate", hiredate.toString());
        message.setFloat("sal", sal);
        message.setFloat("comm", comm);
        message.setInt("deptno", deptno);
```

```
          break;
        case 2: // termination
          message.setInt("empno", empno);
          break;
        case 3: // salary adjustment
          message.setInt("empno", empno);
          message.setFloat("sal", sal);
          break;
        }
        // persistent, no priority or expiration
        qsender.send(message, DeliveryMode.PERSISTENT,
                   0, 0);
      }
      catch (NamingException e) {
        result = false;
        resultMessage = "JNDI error on " + e.getMessage();
      }
      catch (JMSException je) {
        result = false;
        resultMessage = "JMS Error: " + je.getMessage();
      }
      try { qsender.close();     } catch (Exception e) {};
      try { qsession.close();    } catch (Exception e) {};
      try { qconnection.close(); } catch (Exception e) {};
      try { ctx.close();         } catch (Exception e) {};

      return result;
    }
```

You use the same process, but with different object names, to send messages to Topics. See the WebLogic JMS developers guide, *Using WebLogic JMS*, at *http://www.weblogic.com/docs51/classdocs/API_jms.html*, for more about other message types, using header fields and properties, and other message delivery options.

Receiving Messages

Setting up a client to receive JMS messages is nearly the same as setting up to send messages. You look up a ConnectionFactory and a Queue or Topic with JNDI, create a Connection, a Session, and then a message consumer. Then you start the Connection and receive messages.

The `EmpQueueReader.java` example is the Java client application that reads the messages sent by the `EmpTrans` HTTP Servlet. This client application reads a message, displays the transaction it holds, and asks if the transaction should be approved. If the user approves the transaction, the application applies the transaction using the Emp Enterprise JavaBean.

Here is the JMS code from the `EmpQeueuReader.java` example:

Listing 8-2 EmpQueueReader.java

```
public class EmpQueueReader
{
  public final static String JNDI_FACTORY =
               "weblogic.jndi.WLInitialContextFactory";
  public final static String JMS_FACTORY =
               "javax.jms.QueueConnectionFactory";
  public final static String QUEUE =
               "jms.queue.EmpXactQueue";

  static QueueConnectionFactory factory;
  static QueueConnection qconnection;
  static QueueSession qsession;
  static QueueReceiver qreceiver;
  static Queue queue;
  static Context ctx;
  private boolean quit = false;

  /**
   * Create all the necessary objects for receiving
   * messages from a JMS queue.
   */
  public static void main(String[] args)
    throws NamingException, JMSException, RemoteException
  {

    Message message;

    Hashtable ht = new Hashtable();
    ht.put(Context.SECURITY_PRINCIPAL, "guest");
    ht.put(Context.SECURITY_CREDENTIALS, "guest");
    ht.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    ht.put(Context.PROVIDER_URL, "t3://localhost:7001");

    ctx = new InitialContext(ht);
    factory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
    qconnection = factory.createQueueConnection();
    qsession = qconnection.createQueueSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
    queue = (Queue) ctx.lookup(QUEUE);
    qreceiver = qsession.createReceiver(queue);
    qconnection.start();
    while (true) { // get messages until the queue is empty
      message = qreceiver.receiveNoWait();
      if (message == null)
        break;
      if (message instanceof MapMessage) {
        switch (((MapMessage) message).getInt("xactType")) {
        case 1:  // hire
          doHire((MapMessage) message);
          break;
        case 2: // termination
          doTerm((MapMessage) message);
          break;
        case 3: // salary adjustment
          doSal((MapMessage) message);
          break;
        }
      }
    }
    qsession.close();
    qconnection.close();
  }
```

By using the `receiveNoWait()` method in a `while` loop, this example reads and processes all of the messages currently in the Queue and then quits.

Different message types can be mixed on the same Queue or Topic. This example uses `instanceof` to make sure that it has received a `MapMessage`. If another message type happens to get into the Queue, this example ignores it.

The `switch` statement passes the message to a method that handles the transaction type. These methods use the MapMessage `get`*`Type`*`()` methods to retrieve the transaction data from the Message. Then they use the Emp Enterprise JavaBean to add, remove, or update an Employee record. Here is the code that handles hire transactions:

Listing 8-3 EmpQueueReceiver.doHire()

```
  static void doHire(MapMessage message)
    throws JMSException, NamingException, RemoteException
  {
    boolean response;
    Emp e;

    System.out.println("\n\n\n=== Hire request ===");
    System.out.println("Employee name ...... : " +
                       message.getString("ename"));
    System.out.println("Job title .......... : " +
                       message.getString("job"));
    System.out.println("Manager ............ : " +
                       message.getInt("mgr"));
    System.out.println("Hire date .......... : " +
                       message.getString("hiredate"));
    System.out.println("Salary ............. : " +
                       message.getFloat("sal"));
    System.out.println("Commission ......... : " +
                       message.getFloat("comm"));
    System.out.println("Department no ...... : " +
                       message.getInt("deptno"));
    System.out.println("===========================");
    response = Confirm("Approve this request?");
    if (response) {
      try {
       EmpBeanHome home = (EmpBeanHome) ctx.lookup("EmpBeanHome");
        e = home.create(NextEmpno(),
                        message.getString("ename"),
                        message.getString("job"),
                        message.getInt("mgr"),
                        java.sql.Date.valueOf(
                            message.getString("hiredate")),
                        message.getFloat("sal"),
                        message.getFloat("comm"),
                        message.getInt("deptno"));
      }
      catch (CreateException ce) {
        System.out.println("EJB CreateException: " +
               ce.getMessage());
      }
      System.out.println("Created new employee.");
    }
    else
      System.out.println("Ignoring request.");
  }
```

Running the EmpTrans Example

The EmpTrans example has three parts: the Emp Enterprise JavaBean, an HTTP Servlet, and a Java client application. In addition to setting up these separate applications, you must configure WebLogic Server for JMS.

Notes: This example requires the Emp EJB and depends upon the Emp table in the Cloudscape Demo database that is installed with WebLogic Server. If you want to use a different database, you must edit the `EmpTrans.java` and `EmpQueueReader.java` source files. The client application is written to be executed at a command line on the same computer running WebLogic Server. If you want to run the client on a different computer, change the URL in the `EmpQueueReader.java` file before you compile.

Here are steps to set up and run the EmpTrans example:

1. Set up your development environment as described in *Setting your development environment* at *http://www.weblogic.com/docs51/techstart/environment.html*.

2. Set up the Emp Enterprise JavaBean. See "Running the Emp Example" for instructions.

3. Compile the `EmpTrans.java` servlet:

   Windows NT:

   ```
   javac -d %SERVLET_CLASSES% EmpTrans.java
   ```

   UNIX:

   ```
   javac -d $SERVLET_CLASSES EmpTrans.java
   ```

4. Compile the `EmpQueueReader.java` client program:

   Windows NT:

   ```
   javac -d %CLIENT_CLASSES% EmpQueueReader.java
   ```

   UNIX:

   ```
   javac -d $CLIENT_CLASSES EmpQueueReader.java
   ```

5. Add or change the following properties in your `weblogic.properties` file:

   - Register the EmpTrans servlet by adding this property:

     ```
     weblogic.httpd.register.EmpTrans=examples.intro.EmpTrans
     ```

   - Make sure the Emp EJB is deployed:

     ```
     weblogic.ejb.deploy=\
           c:/weblogic/myserver/Emp.jar
     ```

   - Uncomment the following property so that JMS stores persistent messages in the Cloudscape demo database:

     ```
     weblogic.jms.connectionPool=demoPool
     ```

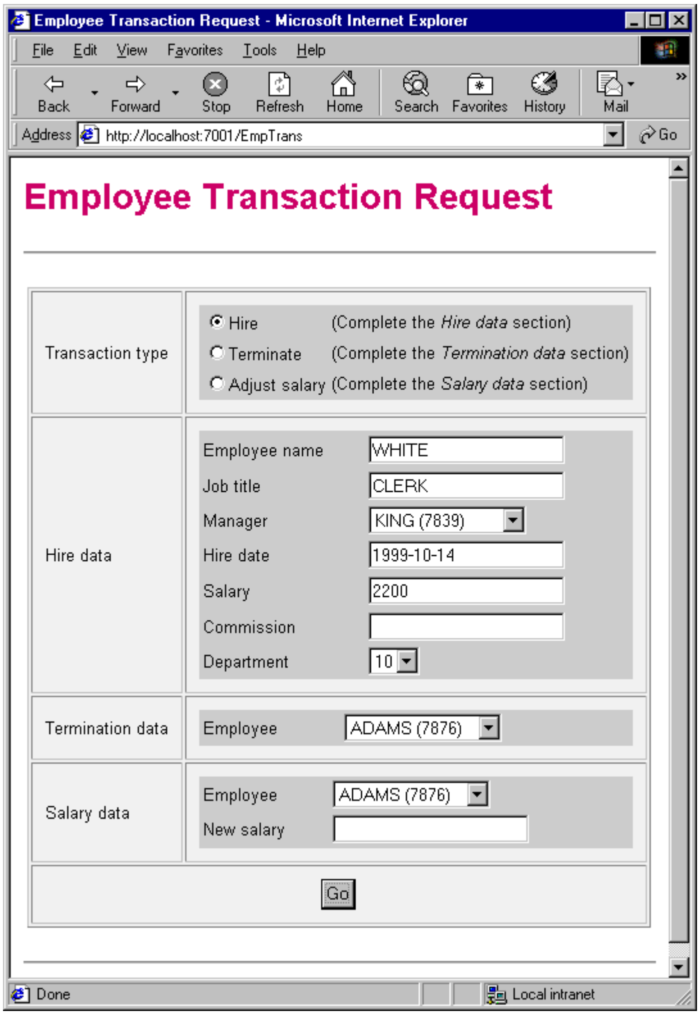
- Define the EmpXactQueue Queue and add ACLs to allow everyone to send or receive messages:

  ```
  weblogic.jms.queue.EmpXactQueue=jms.queue.EmpXactQueue
  weblogic.allow.send.weblogic.jms.queue.EmpXactQueue=everyone
  weblogic.allow.receive.weblogic.jms.queue.EmpXactQueue=\
             everyone
  ```

6. Start WebLogic Server.
7. Load the EmpTrans servlet in a browser with a URL like *http://localhost:7001/EmpTrans*.
8. After entering transactions in the EmpTrans servlet, start the EmpQueueReader client application at a command line with a command such as:

   ```
   java examples.intro.EmpQueueReader
   ```

The EmpTrans servlet displays a web page similar to this:


Employee Transaction Request - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://localhost:7001/EmpTrans
Go
Done
Local intranet

Employee Transaction Request

Transaction type	[✓ Hire] (Complete the *Hire data* section) Terminate (Complete the *Termination data* section) Adjust salary (Complete the *Salary data* section)
Hire data	Employee name: WHITE Job title: CLERK Manager: KING (7839) Hire date: 1999-10-14 Salary: 2200 Commission: Department: 10
Termination data	Employee: ADAMS (7876)
Salary data	Employee: ADAMS (7876) New salary:

Go

The output of the EmpQueueReader application is similar to this:

```
C:\>java examples.intro.EmpQueueReader



=== Hire request ===
Employee name ...... : WHITE
```

```
Job title .......... : CLERK
Manager ............ : 7839
Hire date .......... : 1999-10-14
Salary ............. : 2200.0
Commission ......... : 0.0
Department no ...... : 10
===========================
Approve this request? (yes/no) yes
Created new employee.
```

You can use the SqlServlet servlet to execute a query such as "`select * from emp`" to see that the Emp bean has updated the Emp database table with your transactions.

More about JMS

To find out more about WebLogic JMS, see *Using WebLogic JMS* at *http://www.weblogic.com/docs51/classdocs/API_jms.html*.

Visit the *JMS homepage* at *http://www javasoft.com/products/jms/index.html* for JMS FAQs, tutorials, and other news.

The *javax.jms javadocs* are available online at *http://java.sun.com/products/jms/javadoc-101a/index.html*. You can also download the *JMS specification* from Sun at *http://java.sun.com/products/jms/docs.html*.

Index