BEA WebLogic Server™ and WebLogic Express®

Programming WebLogic JDBC

Release 8.1 Beta
Document Revised: December 9, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JDBC

Part Number	Date	Software Version
N/A	December 9, 2002	BEA WebLogic Server Version 8.1 Beta

Contents

About This Document

1. Introduction to WebLogic JDBC

2. Configuring and Administering WebLogic JDBC

3. Performance Tuning Your JDBC Application

4. Using WebLogic Multitier JDBC Drivers

5. Using Third-Party Drivers with WebLogic Server

6. Testing JDBC Connections and Troubleshooting

About This Document

This document describes how to use JDBC with WebLogic Server™.

The document is organized as follows:

- Chapter 1, "Introduction to WebLogic JDBC," introduces the JDBC components and JDBC API.
- Chapter 2, "Configuring and Administering WebLogic JDBC," describes how to configure JDBC components for use with WebLogic Server Java applications.
- Chapter 3, "Performance Tuning Your JDBC Application," describes how to obtain the best performance from JDBC applications.
- Chapter 4, "Using WebLogic Multitier JDBC Drivers," describes how to set up your WebLogic RMI driver and JDBC clients to use with WebLogic Server.
- Chapter 5, "Using Third-Party Drivers with WebLogic Server," describes how to set up and use third-party drivers with WebLogic Server.
- Chapter 6, "Testing JDBC Connections and Troubleshooting," describes troubleshooting tips when using JDBC with WebLogic Server.

Audience

This document is written for application developers who want to build e-commerce applications using the Java 2 Platform, Enterprise Edition (J2EE) from Sun Microsystems, Inc. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. For more information about JDBC, see the JDBC section on the Sun Microsystems JavaSoft Web site at http://java.sun.com/products/jdbc/index.html.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version your are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and filenames and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Introduction to WebLogic JDBC

The following sections provide an overview of the JDBC components and JDBC API:

Overview of JDBC

Java Database Connectivity (JDBC) is a standard Java API that consists of a set of classes and interfaces written in the Java programming language. Application, tool, and database developers use JDBC to write database applications and execute SQL statements.

JDBC is a *low-level* interface, which means that you use it to invoke (or call) SQL commands directly. In addition, JDBC is a base upon which to build higher-level interfaces and tools, such as Java Message Service (JMS) and Enterprise Java Beans (EJBs).

Using JDBC Drivers with WebLogic Server

JDBC drivers implement the interfaces and classes of the JDBC API. The following sections describe the JDBC driver options that you can use with WebLogic Server.

Types of JDBC Drivers

WebLogic Server uses the following types of JDBC drivers that work in conjunction with each other to provide database access:

- *Two-tier drivers* that provide database access directly between a connection pool and the database. WebLogic Server uses a DBMS vendor-specific JDBC driver, such as the WebLogic jDrivers for Oracle and Microsoft SQL Server, to connect to a back-end database.
- *Multitier drivers* that provide vendor-neutral database access. A Java client application can use a multitier driver to access any database configured in WebLogic server. BEA offers three multitier drivers—RMI, Pool, and JTS. The WebLogic Server system uses these drivers behind the scenes when you use a JNDI look-up to get a connection from a connection pool through a data source.

The middle tier architecture of WebLogic Server, including data sources and connection pools, allows you to manage database resources centrally in WebLogic Server. The vendor-neutral multitier JDBC drivers makes it easier to adapt purchased components to your DBMS environment and to write more portable code.

Table of WebLogic Server JDBC Drivers

The following table summarizes the drivers that WebLogic Server uses.

Table 1-1 JDBC Drivers

Driver Tier	Type and Name of Driver	Database Connectivity	Documentation Sources
Two-tier without support for distributed transactions (non-XA)	Type 2 (requires native libraries): ■ WebLogic jDriver for Oracle ■ Third-party drivers Type 4 (pure Java) ■ WebLogic jDrivers for Microsoft SQL Server ■ Third-party drivers, including: Oracle Thin Sybase jConnect	Between WebLogic Server and DBMS in local transactions.	*Programming WebLogic JDBC* (this document) *Administration Console Online Help*, "Configuring JDBC Connecttion Pools" *Using WebLogic jDriver for Oracle* *Using WebLogic jDriver for Microsoft SQL Server*
Two-tier with support for distributed transactions (XA)	Type 2 (requires native libraries) ■ WebLogic jDriver for Oracle XA	Between WebLogic Server and DBMS in distributed transactions.	*Programming WebLogic JTA* *Administration Console Online Help*, "Configuring JDBC Connecttion Pools" *Using WebLogic jDriver for Oracle*
Multitier	Type 3 ■ WebLogic RMI Driver ■ WebLogic Pool Driver ■ WebLogic JTS (not Type 3)	Between client and WebLogic Server (connection pool). The RMI driver replaces the deprecated t3 driver. The JTS driver is used in distributed transactions. The Pool and JTS drivers are server-side only.	*Programming WebLogic JDBC* (this document)

WebLogic Server JDBC Two-Tier Drivers

The following sections describe Type 2 and Type 4 BEA two-tier drivers used with WebLogic Server to connect to the vendor-specific DBMS.

WebLogic jDriver for Oracle

BEA's WebLogic jDriver for Oracle is included with the WebLogic Server distribution. This driver requires an Oracle client installation. The *WebLogic jDriver for Oracle XA* driver extends the WebLogic jDriver for Oracle for distributed transactions. For additional information, see Using WebLogic jDriver for Oracle at `http://e-docs.bea.com/wls/docs81boracle/index.html`.

WebLogic jDriver for Microsoft SQL Server

BEA's WebLogic jDriver for Microsoft SQL Server, included in the WebLogic Server distribution, is a pure-Java, Type 4 JDBC driver that provides connectivity to Microsoft SQL Server. For more information, see Configuring and Using WebLogic jDriver for MS SQL Server at `http://e-docs.bea.com/wls/docs81b/mssqlserver4/index.html`.

WebLogic Server JDBC Multitier Drivers

The following sections briefly describe the WebLogic multitier JDBC drivers that provide database access to applications. You can use these drivers in server-side applications (also in client applications for the RMI driver), however BEA recommends that you look up a data source from the JNDI tree to get a database connection.

For more details about using these drivers, see Chapter 4, "Using WebLogic Multitier JDBC Drivers."

WebLogic RMI Driver

The WebLogic RMI driver is a multitier, Type 3, Java Database Connectivity (JDBC) driver that runs in WebLogic Server. You can use the WebLogic RMI driver to connect to a database through a connection pool, however, this is not the recommended method. BEA recommends that you look up a data source on the JNDI tree to get a database connection from a connection pool. The data source then internally uses the RMI driver. With either method, the WebLogic RMI driver uses the WebLogic Pool and WebLogic JTS drivers internally to get a connection from a connection pool.

Additionally, when configured in a cluster of WebLogic Servers, the WebLogic RMI driver can be used for clustered JDBC, allowing JDBC clients the benefits of load balancing and failover provided by WebLogic Clusters.

You can use the WebLogic RMI driver with server-side or client applications.

For more details about using the WebLogic RMI driver, see "Using the WebLogic RMI Driver" on page 4-1.

WebLogic Pool Driver

The WebLogic Pool driver enables utilization of connection pools from server-side applications such as HTTP servlets or EJBs. You can use it directly in server-side applications, but BEA recommends that you use a data source through a JNDI look-up to get a connection from a connection pool. Data sources in WebLogic Server use the WebLogic Pool driver internally to get connections from a connection pool.

For information about using the Pool driver, see Accessing Databases in Programming Tasks in *Programming WebLogic HTTP Servlets*.

WebLogic JTS Driver

The WebLogic JTS driver is a multitier JDBC driver that is similar to the WebLogic Pool Driver, but is used in distributed transactions across multiple servers with one database instance. The JTS driver is more efficient than the WebLogic jDriver for Oracle XA driver when working with only one database instance because it avoids two-phase commit. This driver is for use with server-side applications only.

For more details about using the WebLogic JTS driver, see "Using the WebLogic JTS Driver" on page 4-7.

Third-Party Drivers

WebLogic Server works with third-party JDBC drivers that meet the following requirements:

- Are thread-safe.
- Support the JDBC API. Drivers can support extensions to the API, but they must support the JDBC API as a minimum.

- Implement EJB transaction calls in JDBC.

You typically use these drivers when configuring WebLogic Server to create physical database connections in a connection pool.

Sybase jConnect Driver

The two-tier Sybase jConnect Type 4 driver is shipped with your WebLogic Server distribution. You may want to use the latest version of this driver, which is available from the Sybase Web site. For information on using this driver with WebLogic Server, see "Using Third-Party Drivers with WebLogic Server" on page 5-1.

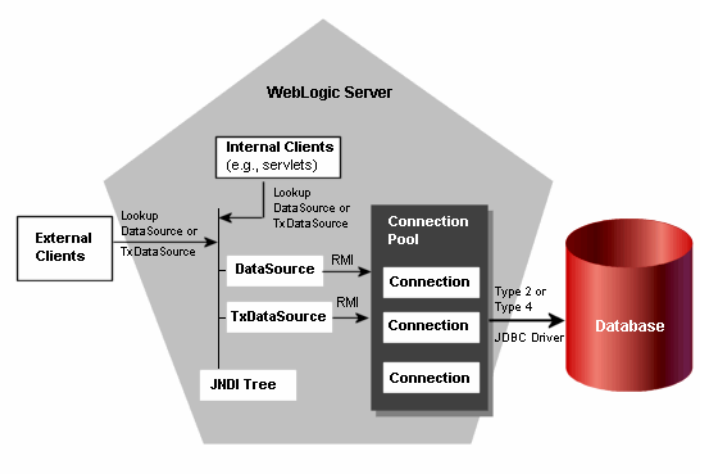
Oracle Thin Driver

The two-tier *Oracle Thin* Type 4 driver bundled with WebLogic Server provides connectivity from WebLogic Server to an Oracle DBMS. You may want to use the latest version of the Oracle Thin driver, which is available from the Oracle Web site. For information on using this driver with WebLogic Server, see "Using Third-Party Drivers with WebLogic Server" on page 5-1.

Overview of Connection Pools

In WebLogic Server, you can configure *connection pools* that provide ready-to-use pools of connections to your DBMS. Client and server-side applications can utilize connections from a connection pool through a DataSource on the JNDI tree (the preferred method) or by using a multitier WebLogic driver. When finished with a connection, applications return the connection to the connection pool.

Figure 1-1 WebLogic Server Connection Pool Architecture



When the connection pool starts up, it creates a specified number of physical database connections. By establishing connections at start-up, the connection pool eliminates the overhead of creating a database connection for each application.

Connection pools require a two-tier JDBC driver to make the physical database connections from WebLogic Server to the DBMS. The two-tier driver can be one of the WebLogic jDrivers or a third-party JDBC driver, such as the Sybase jConnect driver or the Oracle Thin Driver. The following table summarizes the advantages to using connection pools.

Table 1-2 Advantages to Using Connection Pools

Connection Pools Provide These Advantages. . .	With This Functionality . . .
Save time, low overhead	Making a DBMS connection is very slow. With connection pools, connections are already established and available to users. The alternative is for applications to make their own JDBC connections as needed. A DBMS runs faster with dedicated connections than if it has to handle incoming connection attempts at run time.
Manage DBMS users	Allows you to manage the number of concurrent DBMS connections on your system. This is important if you have a licensing limitation for DBMS connections, or a resource concern. Your application does not need to know of or transmit the DBMS username, password, and DBMS location.
Allow use of the DBMS persistence option	If you use the DBMS persistence option with some APIs, such as EJBs, pools are mandatory so that WebLogic Server can control the JDBC connection. This ensures your EJB transactions are committed or rolled back correctly and completely.

This section is an overview of connection pools. For more detailed information, see "Configuring and Using Connection Pools" on page 2-2.

Using Connection Pools with Server-side Applications

For database access from server-side applications, such as HTTP servlets, use a DataSource from the Java Naming and Directory Interface (JNDI) tree or use the WebLogic Pool driver. For two-phase commit transactions, use a TxDataSource from the JNDI tree or use the WebLogic Server JDBC/XA driver, WebLogic jDriver for Oracle/XA. For transactions distributed across multiple servers with one database

instance, use a TxDataSource from the JNDI tree or use the JTS driver. BEA recommends that you access connection pools using the JNDI tree and a DataSource object rather than using WebLogic multitier drivers.

Using Connection Pools with Client-side Applications

BEA offers the RMI driver for client-side, multitier JDBC. The RMI driver provides a standards-based approach using the Java 2 Enterprise Edition (J2EE) specifications. For new deployments, BEA recommends that you use a DataSource from the JNDI tree to access database connections rather than the RMI driver.

The WebLogic RMI driver is a Type 3, multitier JDBC driver that uses RMI and a DataSource object to create database connections. This driver also provides for clustered JDBC, leveraging the load balancing and failover features of WebLogic Server clusters. You can define DataSource objects to enable transactional support or not.

Overview of MultiPools

Relevant only in *single-server* configurations, JDBC MultiPools are "pools of connection pools" that you can set up according to either a high availability or load balancing algorithm. You use a MultiPool in the same manner that you use a connection pool. When an application requests a connection, the MultiPool determines which connection pool will provide a connection, according to the selected algorithm. MultiPools are not supported multiple-server configurations or with distributed transactions.

You can choose one of the following algorithm options for each MultiPool in your WebLogic Server configuration:

- High availability, in which the connection pools are set up as an ordered list and used sequentially.
- Load balancing, in which all listed pools are accessed using a round-robin scheme.

For more information, see "Configuring and Using MultiPools" on page 2-16.

Overview of Clustered JDBC

WebLogic Server allows you to cluster JDBC objects, including data sources, connection pools and MultiPools, to improve the availability of cluster-hosted applications. Each JDBC object you configure for your cluster must exist on *each* managed server in the cluster—when you configure the JDBC objects, target them to the cluster.

For information about JDBC objects in a clustered environment, see "JDBC Connections" in *Using WebLogic Server Clusters* at `http://e-docs.bea.com/wls/docs81b/cluster/overview.html#JDBC`.

Overview of DataSources

Client and server-side JDBC applications can obtain a DBMS connection using a DataSource. A DataSource is an interface between an application and the connection pool. Each data source (such as a DBMS instance) requires a separate DataSource object, which may be implemented as a DataSource class that supports distributed transactions. For more information, see "Configuring and Using DataSources" on page 2-18.

JDBC API

To create a JDBC application, use the *java.sql* API to create the class objects necessary to establish a connection with a data source, to send queries and update statements to the data source, and to process the results. For a complete description of all JDBC interfaces, see the standard JDBC interfaces at `java.sql` Javadoc. Also see the following WebLogic Javadocs:

- weblogic.jdbc.pool

- `weblogic.management.configuration` (MBeans for creating DataSources, connection pools, and MultiPools)

JDBC 2.0

WebLogic Server uses JDK 1.3.1, which supports JDBC 2.0.

Platforms

Supported platforms vary by vendor-specific DBMSs and drivers. For current information, see BEA WebLogic Server Platform Support at `http://e-docs.bea.com/wls/certifications/certifications/index.html`.

2 Configuring and Administering WebLogic JDBC

You use WebLogic Server Administration Console to enable, configure, and monitor features of the WebLogic Server, including JDBC.

The following sections describe how to program the JDBC connectivity components:

For additional information, see

- Administration Console Online Help at `http://e-docs.bea.com/wls/docs81b/ConsoleHelp/index.html`.
- WebLogic Server Javadocs at `http://e-docs.bea.com/wls/docs81b/javadocs/index.html` for the following interfaces:
  - `weblogic.management.configuration.JDBCConnectionPoolMBean`
  - `weblogic.management.configuration.JDBCDataSourceFactoryMBean`
  - `weblogic.management.configuration.JDBCDataSourceMBean`
  - `weblogic.management.configuration.JDBCMultiPoolMBean`

- `weblogic.management.configuration.JDBCTxDataSourceMBean`
- `weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean`

Configuring and Using Connection Pools

A connection pool is a named group of identical JDBC connections to a database that are created when the connection pool is registered, either at WebLogic Server startup or dynamically during run time. Your application "borrows" a connection from the pool, uses it, then returns it to the pool by closing it. Also see "Overview of Connection Pools" on page 1-6.

Advantages to Using Connection Pools

Connection pools provide numerous performance and application design advantages:

- Using connection pools is far more efficient than creating a new connection for each client each time they need to access the database.
- You do not need to hard-code details such as the DBMS password in your application.
- You can limit the number of connections to your DBMS. This can be useful for managing licensing restrictions on the number of connections to your DBMS.
- You can change the DBMS you are using without changing your application code.

The attributes for a configuring a connection pool are defined in the Administration Console Online Help. There is also an API that you can use to programmatically create connection pools in a running WebLogic Server; see "Creating a Connection Pool Dynamically" on page 2-5. You can also use the command line; see the Web Logic Server Command-Line Interface Reference at `http://e-docs.bea.com/wls/docs81b/admin_ref/cli.html`.

Creating a Connection Pool at Startup

To create a startup (static) connection pool, you define attributes and permissions in the Administration Console before starting WebLogic Server. WebLogic Server opens JDBC connections to the database during the startup process and adds the connections to the pool.

To configure a connection pool in the Administration Console, in the navigation tree in the left pane, expand the Services and JDBC nodes, then select Connection Pool. The right pane displays a list of existing connection pools. Click the *Configure a new JDBC Connection Pool* text link to create a connection pool.

For step-by-step instructions and a description of connection pool attributes, see the Administration Console Online Help, available when you click the question mark in the upper-right corner of the Administration Console or at `http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_connection_pools.html`.

Avoiding Server Lockup with the Correct Number of Connections

When your applications attempt to get a connection from a connection pool in which there are no available connections, the connection pool throws an exception stating that a connection is not available in the connection pool. Connection pools do not queue requests for a connection. To avoid this error, make sure your connection pool can expand to the size required to accommodate your peak load of connection requests.

To set the maximum number of connections for a connection pool in the Administration Console, expand the navigation tree in the left pane to show the Services—JDBC—Connection Pools nodes and select a connection pool. Then, in the right pane, select the Configuration—Connections tab and specify a value for `Maximum Capacity`.

Database Passwords in Connection Pool Configuration

When you create a connection pool, you typically include at least one password to connect to the database. If you use an open string to enable XA, you may use two passwords. You can enter the passwords as a name-value pair in the `Properties` field or you can enter them in their respective fields:

- **`Password`**. Use this field to set the database password. This value overrides any `password` value defined in the `Properties` passed to the tier-2 JDBC Driver when creating physical database connections. The value is encrypted in the `config.xml` file (stored as the `Password` attribute in the `JDBCConnectionPool` tag) and is hidden on the administration console.
- **`Open String Password`**. Use this field to set the password in the open string that the transaction manager in WebLogic Server uses to open a database connection. This value overrides any password defined as part of the open string in the `Properties` field. The value is encrypted in the `config.xml` file (stored as the `XAPassword` attribute in the `JDBCConnectionPool` tag) and is hidden on the Administration Console. At runtime, WebLogic Server reconstructs the open string with the password you specify in this field. The open string in the Properties field should follow this format:

  ```
  openString=Oracle_XA+Acc=P/userName/+SesTm=177+DB=demoPool+Thre
  ads=true=Sqlnet=dvi0+logDir=.
  ```

  Note that after the `userName` there is no password.

If you specify a password in the `Properties` field when you first configure the connection pool, WebLogic Server removes the password from the `Properties` string and sets the value as the `Password` value in an encrypted form the next time you start WebLogic Server. If there is already a value for the `Password` attribute for the connection pool, WebLogic Server does not change any values. However, the value for the `Password` attribute overrides the password value in the `Properties` string. The same behavior applies to any password that you define as part of an open string. For example, if you include the following properties when you first configure a connection pool:

```
user=scott;
password=tiger;
openString=Oracle_XA+Acc=p/scott/tiger+SesTm=177+db=jtaXaPool+Thr
eads=true+Sqlnet=lcs817+logDir=.+dbgFl=0x15;server=lcs817
```

The next time you start WebLogic Server, it moves the database password and the password included in the open string to the Password and Open String Password attributes, respectively, and the following value remains for the `Properties` field:

```
user=scott;
openString=Oracle_XA+Acc=p/scott/+SesTm=177+db=jtaXaPool+Threads=
true+Sqlnet=lcs817+logDir=.+dbgFl=0x15;server=lcs817
```

After a value is established for the `Password` or `Open String Password` attributes, the values in these attributes override the respective values in the `Properties` attribute. That is, continuing with the previous example, if you specify `tiger2` as the database password in the `Properties` attribute, WebLogic Server ignores the value and continues to use `tiger` as the database password, which is the current encrypted value of the `Password` attribute. To change the database password, you must change the `Password` attribute.

Note: The value for `Password` and `Open String Password` do not need to be the same.

Permissions

Creating a Connection Pool Dynamically

The `JDBCConnectionPool` administration MBean as part of the WebLogic Server management architecture (JMX). You can use the `JDBCConnectionPool` MBean to create and configure a connection pool dynamically from within a Java application. That is, from your client or server application code, you can create a connection pool in a WebLogic Server that is already running.

You can also use the `CREATE_POOL` command in the WebLogic Server command line interface to dynamically create a connection pool. See CREATE_POOL at `http://e-docs.bea.com/wls/docs81b/admin_ref/cli.html#cli_create_pool`.

To dynamically create a connection pool using the `JDBCConnectionPool` administration MBean, follow these main steps:

1. Import required packages.
2. Look up the administration MBeanHome in the JNDI tree.
3. Get the server MBean.
4. Create the connection pool MBean.
5. Set the properties for the connection pool.
6. Add the target.

7. Create a DataSource object.

Note: Dynamically created connection pools must use dynamically created DataSource objects. For a DataSource to exist, it must be associated with a connection pool. Also, a one-to-one relationship exists between DataSource objects and connection pools in WebLogic Server. Therefore, you must create a DataSource to use with a connection pool.

When you create a connection pool using the `JDBCConnectionPool` MBean, the connection pool is added to the server configuration and will be available even if you stop and restart the server. If you do not want the connection pool to be persistent, you must remove it programmatically.

Also, you can temporarily disable dynamically created connection pools, which suspends communication with the database server through any connection in the pool. When a disabled pool is re-enabled, each connection returns to the same state as when the pool was disabled; clients can continue their database operations exactly where they left off.

For more information about using MBeans to manage WebLogic Server, see Programming WebLogic Management Services with JMX at `http://e-docs.bea.com/wls/docs81b/jmx/index.html`. For more information about the `JDBCConnectionPool` MBean, see the Javadoc at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html`.

Dynamic Connection Pool Sample Code

The following sections show code samples for performing the main steps to create a connection pool dynamically.

Import Packages

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.sql.DataSource;
import weblogic.jndi.Environment;
import weblogic.management.configuration.JDBCConnectionPoolMBean;
import weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean;
import weblogic.management.configuration.JDBCDataSourceMBean;
```

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

Look Up the Administration MBeanHome

```
mbeanHome = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
```

Get the Server MBean

```
serverMBean = (ServerMBean)mbeanHome.getAdminMBean(serverName, "Server");
//Create a WebLogic object name for the Server MBean
//to use to create a name for the JDBCConnectionPoolRuntime MBean.
WebLogicObjectName pname = new WebLogicObjectName("server1", "ServerRuntime",
mbeanHome.getDomainName(),"server1");
//Create a WebLogic object name for the JDBCConnectionPoolRuntime MBean
//to use to create or get the JDBCConnectionPoolRuntime MBean.
WebLogicObjectName oname = new WebLogicObjectName(cpName,
"JDBCConnectionPoolRuntime", mbeanHome.getDomainName(),"server1", pname);
JDBCConnectionPoolRuntimeMBean cprmb =
(JDBCConnectionPoolRuntimeMBean)mbeanHome.getMBean(oname);
```

Create the Connection Pool MBean

```
// Create ConnectionPool MBean
cpMBean = (JDBCConnectionPoolMBean)mbeanHome.createAdminMBean(
  cpName, "JDBCConnectionPool",
  mbeanHome.getDomainName());
```

Set the Connection Pool Properties

```
Properties pros = new Properties();
pros.put("user", "scott");
  pros.put("server", "lcdbnt1");

// Set DataSource attributes
cpMBean.setURL("jdbc:weblogic:oracle");
cpMBean.setDriverName("weblogic.jdbc.oci.Driver");
cpMBean.setProperties(pros);
cpMBean.setPassword("tiger");
cpMBean.setLoginDelaySeconds(1);
cpMBean.setInitialCapacity(1);
cpMBean.setMaxCapacity(10);
cpMBean.setCapacityIncrement(1);
cpMBean.setShrinkingEnabled(true);
```

```
cpMBean.setShrinkPeriodMinutes(10);
cpMBean.setRefreshMinutes(10);
cpMBean.setTestTableName("dual");
```

Note: In this example, the database password is set using the `setPassword(String)` method instead of including it with the user and server names in `Properties`. When you use the `setPassword(String)` method, WebLogic Server encrypts the password in the `config.xml` file and when displayed on the administration console. BEA recommends that you use this method to avoid storing database passwords in clear text in the `config.xml` file.

Add the Target

```
cpMBean.addTarget(serverMBean);
```

Create a DataSource

```
public void createDataSource() throws SQLException {
  try {
    // Get context
    Environment env = new Environment();
    env.setProviderUrl(url);
    env.setSecurityPrincipal(userName);
    env.setSecurityCredentials(password);
    ctx = env.getInitialContext();

      // Create DataSource  MBean
      dsMBeans = (JDBCDataSourceMBean)mbeanHome.createAdminMBean(
        cpName, "JDBCDataSource",
        mbeanHome.getDomainName());

      // Set DataSource attributes
      dsMBeans.setJNDIName(cpJNDIName);
      dsMBeans.setPoolName(cpName);

      // Startup datasource
      dsMBeans.addTarget(serverMBean);

    } catch (Exception ex) {
      ex.printStackTrace();
      throw new SQLException(ex.toString());

    }

  }
```

Removing a Dynamic Connection Pool and DataSource

The following code sample shows how to remove a dynamically created connection pool. If you do not remove dynamically created connection pools, they will remain available even after the server is stopped and restarted.

```
public void deleteConnectionPool() throws SQLException {
  try {
    // Remove dynamically created connection pool from the server
    cpMBean.removeTarget(serverMBean);
    // Remove dynamically created connection pool from the
configuration
    mbeanHome.deleteMBean(cpMBean);
  } catch (Exception ex) {
    throw new SQLException(ex.toString());
  }
}


  public void deleteDataSource() throws SQLException {

    try {

      // Remove dynamically created datasource from the server

      dsMBeans.removeTarget(serverMBean);

     // Remove dynamically created datasource from the configuration

      mbeanHome.deleteMBean(dsMBeans);

    } catch (Exception ex) {

      throw new SQLException(ex.toString());

    }

  }
```

Managing Connection Pools

Note: Many methods described in this section have been updated with the latest release of WebLogic Server. See the related Javadocs for updated information:

- `JDBCConnectionPoolMBean` at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html`
- `JDBCConnectionPoolRuntimeMBean` at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/runtime/JDBCConnectionPoolRuntimeMBean.html`

New methods include:

- `setConnectionReserveTimeoutSeconds(int seconds)`—Sets the number of seconds that a connection request will wait for a connection from a connection pool while blocking other requests on the thread.
- `setHighestNumUnavailable(int count)`—Sets the maximum number of connections in the pool that can be made unavailable (to an application) for purposes like refreshing the connection, etc.
- `setTestConnectionsOnCreate(boolean enable)`—Enables testing of physical database connections when the connection is initialized.
- `setInitTableName(java.lang.String table)`—Sets the name of the table used for testing a physical database connection when the connection is initialized.
- `setStatementCacheSize(int cacheSize)`—Sets the number of Prepared and Callable Statements stored in the cache for further use. WebLogic Server can reuse statements in the cache without reloading them, which can increase server performance. This replaces `setPreparedStatementCacheSize(int cacheSize)`, which is deprecated.
- `setStatementCacheType(java.lang.String type)`—Sets the algorithm type for the statement cache: LRU (least recently used) or Fixed. See "Increasing Performance with the Statement Cache" at `http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_connection_pools.html#statementcache`.
- `clearStatementCache()`—Clears the cache of statements for each connection in the connection pool.
- `setConnectionCreationRetryFrequencySeconds(int seconds)`—Sets the interval between retries for creating physical database connections. If the database is unavailable, Weblogic Server will retry to create database connections after this interval has elapsed.

- `testPool()`—Tests the pool by reserving and releasing a connection from it.

The `JDBCConnectionPool` and `JDBCConnectionPoolRuntime` MBeans provide methods to manage connection pools and obtain information about them. Methods are provided for:

- Retrieving information about a pool
- Disabling a connection pool, which prevents clients from obtaining a connection from it
- Enabling a disabled pool
- Shrinking a pool, which releases unused connections until the pool has reached the minimum specified pool size
- Refreshing a pool, which closes and reopens its connections
- Shutting down a pool

The `JDBCConnectionPool` and `JDBCConnectionPoolRuntime` MBeans replace the `weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool` classes, which are deprecated.

For more information about methods provided by the `JDBCConnectionPoolMBean`, see the Javadoc at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html`. For more information about the methods provided by the `JDBCConnectionPoolRuntimeMBean`, see the Javadoc at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/runtime/JDBCConnectionPoolRuntimeMBean.html`.

Retrieving Information About a Pool

```
boolean x = JDBCConnectionPoolRuntimeMBean.poolExists(cpName);

props = JDBCConnectionPoolRuntimeMBean.getProperties();
```

The `poolExists()` method tests whether a connection pool with a specified name exists in the WebLogic Server. You can use this method to determine whether a dynamic connection pool has already been created or to ensure that you select a unique name for a dynamic connection pool you want to create.

The `getProperties()` method retrieves the properties for a connection pool.

Disabling a Connection Pool

Need to update this section

`JDBCConnectionPoolRuntimeMBean.disableDroppingUsers()`

`JDBCConnectionPoolRuntimeMBean.disableFreezingUsers()`

`JDBCConnectionPoolRuntimeMBean.enable()`

You can temporarily disable a connection pool, preventing any clients from obtaining a connection from the pool. Only the "system" user or users granted "admin" permission by an ACL associated with a connection pool can disable or enable the pool.

After you call `disableFreezingUsers()`, clients that currently have a connection from the pool are suspended. Attempts to communicate with the database server throw an exception. Clients can, however, close their connections while the connection pool is disabled; the connections are then returned to the pool and cannot be reserved by another client until the pool is enabled.

Use `disableDroppingUsers()` to not only disable the connection pool, but to destroy the client's JDBC connection to the pool. Any transaction on the connection is rolled back and the connection is returned to the connection pool. The client's JDBC connection context is no longer valid.

When a pool is enabled after it has been disabled with `disableFreezingUsers()`, the JDBC connection states for each in-use connection are exactly as they were when the connection pool was disabled; clients can continue JDBC operations exactly where they left off.

You can also use the `disable_pool` and `enable_pool` commands of the `weblogic.Admin` class to disable and enable a pool.

Shrinking a Connection Pool

`JDBCConnectionPoolRuntimeMBean.shrink()`

A connection pool has a set of properties that define the initial and maximum number of connections in the pool (`initialCapacity` and `maxCapacity`), and the number of connections added to the pool when all connections are in use (`capacityIncrement`). When the pool reaches its maximum capacity, the maximum number of connections are opened, and they remain opened unless you shrink the pool.

You may want to drop some connections from the connection pool when a peak usage period has ended, freeing up WebLogic Server and DBMS resources.

Shutting Down a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.shutdownSoft()
```

```
JDBCConnectionPoolRuntimeMBean.shutdownHard()
```

These methods destroy a connection pool. Connections are closed and removed from the pool and the pool dies when it has no remaining connections. Only the "system" user or users granted "admin" permission by an ACL associated with a connection pool can destroy the pool.

The `shutdownSoft()` method waits for connections to be returned to the pool before closing them.

The `shutdownHard()` method kills all connections immediately. Clients using connections from the pool get exceptions if they attempt to use a connection after `shutdownHard()` is called.

You can also use the `destroy_pool` command of the `weblogic.Admin` class to destroy a pool.

Resetting a Pool

```
JDBCConnectionPoolRuntimeMBean.reset()
```

You can configure a connection pool to test its connections either periodically, or every time a connection is reserved or released. Allowing the WebLogic Server to automatically maintain the integrity of pool connections should prevent most DBMS connection problems. In addition, WebLogic provides methods you can call from an application to refresh all connections in the pool or a single connection you have reserved from the pool.

The `JDBCConnectionPoolRuntimeMBean.reset()` method closes and reopens all allocated connections in a connection pool. This may be necessary after the DBMS has been restarted, for example. Often when one connection in a connection pool has failed, all of the connections in the pool are bad.

Use any of the following means to reset a connection pool:

- The Administration Console.

- The `weblogic.Admin` command (as a user with administrative privileges) to reset a connection pool, as an administrator. Here is the pattern:

```
$ java weblogic.Admin WebLogicURL RESET_POOL poolName system passwd
```

  You might use this method from the command line on an infrequent basis. There are more efficient programmatic ways that are also discussed here.

- The `reset()` method from the `JDBCConnectionPoolRuntimeMBean` in your client application.

  The last case requires the most work for you, but also gives you flexibility. To reset a pool using the `reset()` method:

  a. In a `try` block, test a connection from the connection pool with a SQL statement that is guaranteed to succeed under any circumstances so long as there is a working connection to the DBMS. An example is the SQL statement `select 1 from dual` which is guaranteed to succeed for an Oracle DBMS.

  b. Catch the `SQLException`.

  c. Call the `reset()` method in the catch block.

Using weblogic.jdbc.common.JdbcServices and weblogic.jdbc.common.Pool Classes (Deprecated)

Previous versions of WebLogic Server included classes that you could use to programmatically create and manage connection pools: `weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool`. These classes are now deprecated. Although these classes are still available, BEA recommends that you use the `JDBCConnectionPool` MBean instead of these classes to dynamically create and manage connection pools.

When you use the `JDBCConnectionPool` MBean to create or modify a connection pool on a managed server, the JMX service immediately notifies the administration server of the change. When you use `weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool` to create or modify a connection pool, the following actions are *not* conveyed to the Administration Server:

- `shutdown`
- `retrieve`

- `refresh`
- `enable`
- `disable`

After any of these actions, applications on managed servers that use the affected connection pool may fail.

For more information about `weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool`, see "Configuring WebLogic JDBC Features" in *Programming WebLogic JDBC* for WebLogic Server 6.1 at `http://edocs.bea.com/wls/docs61/jdbc/programming.html`.

Application-Scoped JDBC Connection Pools

When you package your enterprise applications, you can include the `weblogic-application.xml` supplemental deployment descriptor, which you use to configure *application scoping*. Within the `weblogic-application.xml` file, you can configure JDBC connection pools that are created when you deploy the enterprise application.

An instance of the connection pool is created with each instance of your application. This means an instance of the pool is created with the application on each node that the application is targeted to. It is important to keep this in mind when considering pool sizing.

Connection pools created in this manner are known as *application-scoped connection pools*, *app scoped pools*, *application local pools*, *app local pools*, or *local pools*, and are scoped for the enterprise application only. That is, they are isolated for use by the enterprise application.

For more information about application scoping and application scoped resources, see:

- weblogic-application.xml Deployment Descriptor Elements in *Developing WebLogic Server Applications* at `http://e-docs.bea.com/wls/docs81b/programming/app_xml.html#app-scoped-pool`.
- Packaging Enterprise Applications in *Developing WebLogic Server Applications* at

http://e-docs.bea.com/wls/docs81b/programming/packaging.html#pack009.

- Two-Phase Deployment Protocol in *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81b/deployment/concepts.html#two_phase.

Configuring and Using MultiPools

A MultiPool is a "pool of pools." MultiPools contain a configurable algorithm for choosing which connection pool will return a connection to the client.

You create a MultiPool by first creating connection pools, then creating the MultiPool using the Administration Console or WebLogic Management API and assigning the connection pools to the MultiPool.

For more information about MultiPools, see the Administration Console Online Help at http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_multipools.html. For information about the `JDBCMultiPoolMBean`, see the WebLogic Server Javadocs at http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/management/configuration/JDBCMultiPoolMBean.html.

MultiPool Features

A MultiPools is a pool of connection pools in a single server. All the connections in a particular *connection pool* are created identically with a single database, single user, and the same connection attributes; that is, they are attached to a single database. However, the connection pools within a *MultiPool* may be associated with different users or DBMSs.

MultiPools are used in local transactions and are not supported by WebLogic Server for distributed transactions.

Choosing the MultiPool Algorithm

Before you set up a MultiPool, you need to determine the primary purpose of the MultiPool—high availability or load balancing. You can choose the algorithm that corresponds with your requirements.

Note: Capacity is not a failover reason, because users have the right to set capacity. MultiPools take effect only if loss of database connectivity has occurred.

High Availability

The High Availability algorithm provides an ordered list of connection pools. Normally, every connection request to this kind of MultiPool is served by the first pool in the list. If a database connection via that pool fails, then a connection is sought sequentially from the next pool on the list.

Load Balancing

Connection requests to a load balancing MultiPool are served from any connection pool in the list. Pools are added without any attached ordering and are accessed using a round-robin scheme. When switching connections, the connection pool just after the last pool accessed is selected.

Guidelines to Setting Wait for Connection Times

Setting wait for connection times is a property of the connection attempt. If you are familiar with setting waiting time to pool connections, the wait for connection property applies to every connection tapped in a given connection attempt.

You can add any connection pool to a MultiPool. However, you optimize your resources depending on how you set the *wait for connection* time when you configure your connection pools.

Messages and Error Conditions

Users may request information regarding the connection pool from which the connection originated.

Exceptions

Entries are posted to the JDBC log under these circumstances:

- At boot time, when a connection pool is added to a MultiPool.
- Whenever there is a switch to a new connection pool within the MultiPool, either during load balancing or high availability.

Capacity Issues

In a high availability scenario, the fact that the first pool in the list is busy (all connections are being used) does not trigger an attempt to get a connection from the next pool in the list.

Configuring and Using DataSources

As with Connection Pools and MultiPools, you can create DataSource objects in the Administration Console or using the WebLogic Management API. DataSource objects can be defined with or without transaction services. You configure connection pools and MultiPools before you define the pool name attribute for a DataSource.

DataSource objects, along with the JNDI, provide access to connection pools for database connectivity. Each DataSource can refer to one connection pool or MultiPool. However, you can define multiple DataSources that use a single connection pool. This allows you to define both transaction and non-transaction-enabled DataSource objects that share the same database.

WebLogic Server supports two types of DataSource objects:

- DataSources (for local transactions only)

- TxDataSources (for distributed transactions)

If your application meets any of the following criteria, you should use a TxDataSource in WebLogic Server:

- Uses the Java Transaction API (JTA)
- Uses the WebLogic Server EJB container to manage transactions
- Includes multiple database updates during a single transaction.

If you want applications to use a DataSource to get a database connection from a connection pool (the preferred method), you should define the DataSource in the Administration Console before running your application. For more information about how to configure a DataSource and when to use a TxDataSource, see JDBC DataSources in the Administration Console Online Help at `http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_datasources.html`.

Importing Packages to Access DataSource Objects

To use the DataSource objects in your applications, import the following classes in your client code:

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

Obtaining a Client Connection Using a DataSource

To obtain a connection from a JDBC client, use a Java Naming and Directory Interface (JDNI) lookup to locate the DataSource object, as shown in this code fragment:

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");
```

```
  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myJtsDataSource");
   java.sql.Connection conn = ds.getConnection();

// You can now use the conn object to create
// Statements and retrieve result sets:

   Statement stmt = conn.createStatement();
   stmt.execute("select * from someTable");
   ResultSet rs = stmt.getResultSet();

// Close the statement and connection objects when you are finished:

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
    // a failure occurred
  }
  finally {
    try {ctx.close();}
    catch (Exception e) {
      // a failure occurred
    }
  }
```

(Substitute the correct `hostname` and `port` number for your WebLogic Server.)

Note: The code above uses one of several available procedures for obtaining a JNDI context. For more information on JNDI, see Programming WebLogic JNDI at `http://e-docs.bea.com/wls/docs81b/jndi/index.html`.

Code Examples

See the DataSource code example in the `samples/examples/jdbc/datasource` directory of your WebLogic Server installation.

JDBC Data Source Factories

In WebLogic Server, you can bind a JDBC DataSource resource into the WebLogic Server JNDI tree as a resource factory. You can then map a resource factory reference in the EJB deployment descriptor to an available resource factory in a running WebLogic Server to get a connection from a connection pool.

For details about creating and using a JDBC Data Source factory, see Resource Factories in *Programming WebLogic Enterprise JavaBeans* at `http://e-docs.bea.com/wls/docs81b/ejb/EJB_environment.html#resourcefact`.

CHAPTER

3 Performance Tuning Your JDBC Application

The following sections explain how to get the most out of your applications:

Overview of JDBC Performance

The underlying concepts in Java, JDBC, and DBMS processing are new to many programmers. As Java becomes more widely used, database access and database applications will become increasingly easy to implement. This document provides some tips on how to obtain the best performance from JDBC applications.

WebLogic Performance-Enhancing Features

WebLogic has several features that enhance performance for JDBC applications.

How Connection Pools Enhance Performance

Establishing a JDBC connection with a DBMS can be very slow. If your application requires database connections that are repeatedly opened and closed, this can become a significant performance issue. WebLogic connection pools offer an efficient solution to this problem.

When WebLogic Server starts, connections from the connection pools are opened and are available to all clients. When a client closes a connection from a connection pool, the connection is returned to the pool and becomes available for other clients; the connection itself is not closed. There is little cost to opening and closing pool connections.

How many connections should you create in the pool? A connection pool can grow and shrink according to configured parameters, between a minimum and a maximum number of connections. The best performance will always be when the connection pool has as many connections as there are concurrent users.

Caching Statements and Data

DBMS access uses considerable resources. If your program reuses prepared or callable statements or accesses frequently used data that can be shared among applications or can persist between connections, you can cache prepared statements or data by using the following:

- Statement Cache for a connection pool (`http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_connection_pools.html#statementcache`)
- Read-Only Entity Beans (`http://e-docs.bea.com/wls/docs81b/ejb/EJB_environment.html`)
- JNDI in a Clustered Environment (`http://e-docs.bea.com/wls/docs81b/jndi/jndi.html`)

Designing Your Application for Best Performance

Most performance gains or losses in a database application is not determined by the application language, but by how the application is designed. The number and location of clients, size and structure of DBMS tables and indexes, and the number and types of queries all affect application performance.

The following are general hints that apply to all DBMSs. It is also important to be familiar with the performance documentation of the specific DBMS that you use in your application.

1. Process as Much Data as Possible Inside the Database

Most serious performance problems in DBMS applications come from moving raw data around needlessly, whether it is across the network or just in and out of cache in the DBMS. A good method for minimizing this waste is to put your logic where the data is—in the DBMS, not in the client —even if the client is running on the same box as the DBMS. In fact, for some DBMSs a fat client and a fat DBMS sharing one CPU is a performance disaster.

Most DBMSs provide stored procedures, an ideal tool for putting your logic where your data is. There is a significant difference in performance between a client that calls a stored procedure to update 10 rows, and another client that fetches those rows, alters them, and sends update statements to save the changes to the DBMS.

Also review the DBMS documentation on managing cache memory in the DBMS. Some DBMSs (Sybase, for example) provide the means to partition the virtual memory allotted to the DBMS, and to guarantee certain objects exclusive use of some fixed areas of cache. This means that an important table or index can be read once from disk and remain available to all clients without having to access the disk again.

2. Use Built-in DBMS Set-based Processing

SQL is a set processing language. DBMSs are designed from the ground up to do set-based processing. Accessing a database one row at a time is, without exception, slower than set-based processing and, on some DBMSs is poorly implemented. For example, it will always be faster to update each of four tables one at a time for all the 100 employees represented in the tables than to alter each table 100 times, once for each employee.

Many complicated processes that were originally thought too complex to do any other way but row-at-a-time have been rewritten using set-based processing, resulting in improved performance. For example, a major payroll application was converted from a huge slow COBOL application to four stored procedures running in series, and what took hours on a multi-CPU machine now takes fifteen minutes with many fewer resources used.

3. Make Your Queries Smart

Frequently customers ask how to tell how many rows will be coming back in a given result set. The only way to find out without fetching all the rows is by issuing the same query using the *count keyword:*

```
SELECT count(*) from myTable, yourTable where ...
```

This returns the number of rows the original query would have returned, assuming no change in relevant data. The actual count may change when the query is issued if other DBMS activity has occurred that alters the relevant data.

Be aware, however, that this is a resource-intensive operation. Depending on the original query, the DBMS may perform nearly as much work to count the rows as it will to send them.

Make your application queries as specific as possible about what data it actually wants. For example, tailor your application to select into temporary tables, returning only the count, and then sending a refined second query to return only a subset of the rows in the temporary table.

Learning to select only the data you really want at the client is crucial. Some applications ported from ISAM (a pre-relational database architecture) will unnecessarily send a query selecting all the rows in a table when only the first few rows are required. Some applications use a 'sort by' clause to get the rows they want to come back first. Database queries like this cause unnecessary degradation of performance.

Proper use of SQL can avoid these performance problems. For example, if you only want data about the top three earners on the payroll, the proper way to make this query is with a correlated subquery. Table 3-1 shows the entire table returned by the SQL statement

```
select * from payroll
```

Table 3-1 Full Results Returned

Name	Salary
Joe	10
Mikes	20
Sam	30
Tom	40
Jan	50
Ann	60
Sue	70
Hal	80
May	80

A correlated subquery

```
select p.name, p.salary from payroll p
where 3 >= (select count(*) from payroll pp
where pp.salary >= p.salary);
```

returns a much smaller result, shown in Table 3-2.

Table 3-2 Results from Subquery

Name	Salary
Sue	70
Hal	80
May	80

This query returns only *three rows, with the name and salary of the top three earners.* It scans through the payroll table, and for every row, it goes through the whole payroll table again in an inner loop to see how many salaries are higher than the current row of the outer scan. This may look complicated, but DBMSs are designed to use SQL efficiently for this type of operation.

4. Make Transactions Single-batch

Whenever possible, collect a set of data operations and submit an update transaction in one statement in the form:

```
BEGIN TRANSACTION
     UPDATE TABLE1...
     INSERT INTO TABLE2
     DELETE TABLE3
COMMIT
```

This approach results in better performance than using separate statements and commits. Even with conditional logic and temporary tables in the batch, it is preferable because the DBMS obtains all the locks necessary on the various rows and tables, and uses and releases them in one step. Using separate statements and commits results in many more client-to-DBMS transmissions and holds the locks in the DBMS for much longer. These locks will block out other clients from accessing this data, and, depending on whether different updates can alter tables in different orders, may cause deadlocks.

Warning: If any individual statement in the preceding transaction fails, due, for instance, to violating a unique key constraint, you should put in conditional SQL logic to detect statement failure and to roll back the transaction rather than commit. If, in the preceding example, the insert failed, most DBMSs return an error message about the failed insert, but behave as if you got the message between the second and third statement, and decided to commit anyway! Microsoft SQL Server offers a connection option enabled by executing the SQL `set xact_abort on`, which automatically rolls back the transaction if any statement fails.

5. Never Have a DBMS Transaction Span User Input

If an application sends a `'BEGIN TRAN'` and some SQL that locks rows or tables for an update, do not write your application so that it must wait on the user to press a key before committing the transaction. That user may go to lunch first and lock up a whole DBMS table until the user returns.

If you require user input to form or complete a transaction, use optimistic locking. Briefly, optimistic locking employs timestamps and triggers in queries and updates. Queries select data with timestamp values and prepare a transaction based on that data, without locking the data in a transaction.

When an update transaction is finally defined by the user input, it is sent as a single submission that includes timestamped safeguards to make sure the data is the same as originally fetched. A successful transaction automatically updates the relevant timestamps for changed data. If an interceding update from another client has altered data on which the current transaction is based, the timestamps change, and the current transaction is rejected. Most of the time, no relevant data has been changed so transactions usually succeed. When a transaction fails, the application can refetch the updated data to present to the user to reform the transaction if desired.

6. Use In-place Updates

Changing a data row in place is much faster than moving a row, which may be required if the update requires more space than the table design can accommodate. If you design your rows to have the space they need initially, updates will be faster, although the table may require more disk space. Because disk space is cheap, using a little more of it can be a worthwhile investment to improve performance.

7. Keep Operational Data Sets Small

Some applications store operational data in the same table as historical data. Over time and with accumulation of this historical data, all operational queries have to read through lots of useless (on a day-to-day basis) data to get to the more current data. Move non-current data to other tables and do joins to these tables for the rarer historical queries. If this can't be done, index and cluster your table so that the most frequently used data is logically and physically localized.

8. Use Pipelining and Parallelism

DBMSs are designed to work best when very busy with lots of different things to do. The worst way to use a DBMS is as dumb file storage for one big single-threaded application. If you can design your application and data to support lots of parallel processes working on easily distinguished subsets of the work, your application will be much faster. If there are multiple steps to processing, try to design your application so that subsequent steps can start working on the portion of data that any prior process has finished, instead of having to wait until the prior process is complete. This may not always be possible, but you can dramatically improve performance by designing your program with this in mind.

CHAPTER

4 Using WebLogic Multitier JDBC Drivers

BEA recommends that you use DataSource objects to get database connections in new applications. DataSource objects, along with the JNDI, provide access to connection pools for database connectivity. For existing or legacy applications that use the JDBC 1.x API, you can use the WebLogic multitier drivers to get database connectivity.

The following sections describe how to use multitier JDBC drivers with WebLogic Server:

Using the WebLogic RMI Driver

The WebLogic RMI driver is a multitier Type 3 JDBC driver WebLogic Server uses to pass database connections from a connection pool to a DataSource or TxDataSource. The DataSource object provides access to database connections for applications through the WebLogic RMI driver. The database connection parameters are set in the connection pool using the Administration Console or the WebLogic Management API, including the two-tier JDBC driver used to access the DBMS. See Figure 1-1.

RMI driver clients make their connection to the DBMS by looking up the DataSource object. This lookup is accomplished by using a Java Naming and Directory Service (JNDI) lookup, or by directly calling WebLogic Server which performs the JNDI lookup on behalf of the client.

The RMI driver replaces the functionality of both the WebLogic t3 driver (deprecated) and the Pool driver, and uses the Java standard Remote Method Invocation (RMI) to connect to WebLogic Server rather than the proprietary t3 protocol.

Because the details of the RMI implementation are taken care of automatically by the driver, a knowledge of RMI is not required to use the WebLogic JDBC/RMI driver.

Setting Up WebLogic Server to Use the WebLogic RMI Driver

The RMI driver is accessible only through DataSource objects, which are created in the Administration Console. You must create DataSource objects in your WebLogic Server configuration before you can use the RMI driver in your applications. For instructions to create a DataSource, see the Administration Console Online Help at `http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_datasources.html#data_source_create`.

Sample Client Code for Using the RMI Driver

The following code samples show how to use the RMI driver to get and use a database connection from a WebLogic Server connection pool.

Import the Required Packages

Before you can use the RMI driver to get and use a database connection, you must import the following packages:

```
javax.sql.DataSource
java.sql.*
java.util.*
javax.naming.*
```

Get the Database Connection

The WebLogic JDBC/RMI client obtains its connection to a DBMS from the DataSource object that you defined in the Administration Console. There are two ways the client can obtain a DataSource object:

- Using a JNDI lookup. This is the preferred and most direct procedure.
- Passing the DataSource name to the RMI driver with the `Driver.connect()`method. In this case, WebLogic Server performs the JNDI look up on behalf of the client.

Using a JNDI Lookup to Obtain the Connection

To access the WebLogic RMI driver using JNDI, obtain a context from the JNDI tree by looking up the name of your DataSource object. For example, to access a DataSource called "`myDataSource`" that is defined in Administration Console:

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
   java.sql.Connection conn = ds.getConnection();

   // You can now use the conn object to create
   //  a Statement object to execute
   //  SQL statements and process result sets:

   Statement stmt = conn.createStatement();
   stmt.execute("select * from someTable");
   ResultSet rs = stmt.getResultSet();

   // Do not forget to close the statement and connection objects
   //  when you are finished:

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
    // a failure occurred
```

```
}
finally {
  try {ctx.close();}
  catch (Exception e) {
    // a failure occurred
  }
}
```

(Where *hostname* is the name of the machine running your WebLogic Server and *port* is the port number where that machine is listening for connection requests.)

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see *Programming WebLogic JNDI* at http://e-docs.bea.com/wls/docs81b/jndi/index.html.

Notice that the JNDI lookup is wrapped in a `try/catch` block in order to catch a failed look up and also that the context is closed in a `finally` block.

Using Only the WebLogic RMI Driver to Obtain a Database Connection

Instead of looking up a DataSource object to get a database connection, you can access WebLogic Server using the `Driver.connect()` method, in which case the JDBC/RMI driver performs the JNDI lookup. To access the WebLogic Server, pass the parameters defining the URL of your WebLogic Server and the name of the DataSource object to the `Driver.connect()` method. For example, to access a DataSource called "myDataSource" as defined in the Administration Console:

```
java.sql.Driver myDriver = (java.sql.Driver)
  Class.forName("weblogic.jdbc.rmi.Driver").newInstance();

String url ="jdbc:weblogic:rmi";

java.util.Properties props = new java.util.Properties();
props.put("weblogic.server.url", "t3://hostname:port");
props.put("weblogic.jdbc.datasource", "myDataSource");

java.sql.Connection conn = myDriver.connect(url, props);
```

(Where *hostname* is the name of the machine running your WebLogic Server and *port* is the port number where that machine is listening for connection requests.)

You can also define the following properties which will be used to set the JNDI user information:

- `weblogic.user`—specifies a username

- `weblogic.credential`—specifies the password for the `weblogic.user`.

Row Caching with the WebLogic RMI Driver

Row caching is a WebLogic Server JDBC feature that improves the performance of your application. Normally, when a client calls `ResultSet.next()`, WebLogic Server fetches a single row from the DBMS and transmits it to the client JVM. With row caching enabled, a single call to `ResultSet.next()` retrieves multiple DBMS rows, and caches them in client memory. By reducing the number of trips across the wire to retrieve data, row caching improves performance.

Note: WebLogic Server will not perform row caching when the client and WebLogic Server are in the same JVM.

You can enable and disable row caching and set the number of rows fetched per `ResultSet.next()` call with the Data Source attributes Row Prefetch Enabled and Row Prefetch Size, respectively. You set Data Source attributes via the Administration Console. To enable row caching and to set the row prefetch size attribute for a DataSource or TxDataSource, follow these steps:

1. In the left pane of the Administration Console, navigate to Services→JDBC→Data Sources or Tx Data Sources, then select the DataSource or TxDataSource for which you want to enable row caching.
2. In the right pane of the Administration Console, select the Configuration tab if it is not already selected.
3. Select the Row Prefetch Enabled check box.
4. In Row Prefetch Size, type the number of rows you want to cache for each `ResultSet.next()` call.

Important Limitations for Row Caching with the WebLogic RMI Driver

Keep the following limitations in mind if you intend to implement row caching with the RMI driver:

- WebLogic Server only performs row caching if the result set type is both TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.

- Certain data types in a result set may disable caching for that result set. These include the following:
  - LONGVARCHAR/LONGVARBINARY
  - NULL
  - BLOB/CLOB
  - ARRAY
  - REF
  - STRUCT
  - JAVA_OBJECT
- Certain ResultSet methods are not supported if row caching is enabled and active for that result set. Most pertain to streaming data, scrollable result sets or data types not supported for row caching. These include the following:
  - `getAsciiStream()`
  - `getUnicodeStream()`
  - `getBinaryStream()`
  - `getCharacterStream()`
  - `isBeforeLast()`
  - `isAfterLast()`
  - `isFirst()`
  - `isLast()`
  - `getRow()`
  - `getObject (Map)`
  - `getRef()`
  - `getBlob()/getClob()`
  - `getArray()`
  - `getDate()`
  - `getTime()`
  - `getTimestamp()`

Using the WebLogic JTS Driver

The Java Transaction Services or JTS driver is a server-side Java Database Connectivity (JDBC) driver that provides access to both connection pools and SQL transactions from applications running in WebLogic Server. Connections to a database are made from a connection pool and use a two-tier JDBC driver running in WebLogic Server to connect to the Database Management System (DBMS) on behalf of your application.

Once a transaction begins, all database operations in an execute thread that get their connection from the *same connection pool* share the *same connection* from that pool. These operations can be made through services such as Enterprise JavaBeans (EJB), or Java Messaging Service (JMS), or by directly sending SQL statements using standard JDBC calls. All of these operations will, by default, share the same connection and participate in the same transaction.When the transaction is committed or rolled back, the connection is returned to the pool.

Although Java clients may not register the JTS driver themselves, they may participate in transactions via Remote Method Invocation (RMI). You can begin a transaction in a thread on a client and then have the client call a remote RMI object. The database operations executed by the remote object become part of the transaction that was begun on the client. When the remote object is returned back to the calling client, you can then commit or roll back the transaction. The database operations executed by the remote objects must all use the same connection pool to be part of the same transaction.

Sample Client Code for Using the JTS Driver

To use the JTS driver, you must first use the Administration Console to create a connection pool in WebLogic Server. For more information, see "Configuring and Using Connection Pools" on page 2-2.

This explanation demonstrates creating and using a JTS transaction from a server-side application and uses a connection pool named "`myConnectionPool`."

1. Import the following classes:

```
import javax.transaction.UserTransaction;
import java.sql.*;
```

```
import javax.naming.*;
import java.util.*;
import weblogic.jndi.*;
```

2. Establish the transaction by using the `UserTransaction` class. You can look up this class on the JNDI tree. The `UserTransaction` class controls the transaction on the current execute thread. Note that this class does not represent the transaction itself. The actual context for the transaction is associated with the current execute thread.

```
Context ctx = null;
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");

// Parameters for the WebLogic Server.
// Substitute the correct hostname, port number
// user name, and password for your environment:
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.SECURITY_PRINCIPAL, “Fred”);
env.put(Context.SECURITY_CREDENTIALS, “secret”);

ctx = new InitialContext(env);

UserTransaction tx = (UserTransaction)
  ctx.lookup("javax.transaction.UserTransaction");
```

3. Start a transaction on the current thread:

```
tx.begin();
```

4. Load the JTS driver:

```
Driver myDriver = (Driver)
 Class.forName("weblogic.jdbc.jts.Driver").newInstance();
```

5. Get a connection from the connection pool:

```
Properties props = new Properties();
props.put("connectionPoolID", "myConnectionPool");

conn = myDriver.connect("jdbc:weblogic:jts", props);
```

6. Execute your database operations. These operations may be made by any service that uses a database connection, including EJB, JMS, and standard JDBC statements. These operations must use the JTS driver to access the same connection pool as the transaction begun in step 3 in order to participate in that transaction.

If the additional database operations using the JTS driver use a *different connection pool* than the one specified in step 5, an exception will be thrown when you try to commit or roll back the transaction.

7. Close your connection objects. Note that closing the connections does not commit the transaction nor return the connection to the pool:

   ```
   conn.close();
   ```

8. Execute any other database operations. If these operations are made by connecting to the same connection pool, the operations will use the same connection from the pool and become part of the same `UserTransaction` as all of the other operations in this thread.

9. Complete the transaction by either committing the transaction or rolling it back. In the case of a commit, the JTS driver commits all the transactions on all connection objects in the current thread and returns the connection to the pool.

   ```
   tx.commit();

   // or:

   tx.rollback();
   ```

Using the WebLogic Pool Driver

The WebLogic Pool driver enables utilization of connection pools from server-side applications such as HTTP servlets or EJBs. For information about using the Pool driver, see "Accessing Databases" in Programming Tasks in *Programming WebLogic HTTP Servlets*.

CHAPTER

5 Using Third-Party Drivers with WebLogic Server

The following sections describe how to set up and use third-party JDBC drivers:

Overview of Third-Party JDBC Drivers

WebLogic Server works with third-party JDBC drivers that offer the following functionality:

- Are thread-safe

- Can implement transactions using standard JDBC statements

This section describes how to set up and use the following third-party JDBC drivers with WebLogic Server:

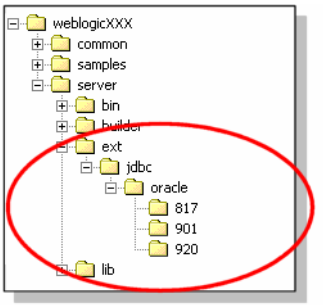
- Oracle Thin Driver 8.1.7, 9.0.1, or 9.2.0 (included in WebLogic Server installation)
- Sybase jConnect Driver 4.5 and 5.5
- IBM Informix JDBC Driver
- Microsoft SQL Server Driver for JDBC

WebLogic Server includes three versions of the Oracle Thin Driver.

The 9.2.0 version of the Oracle Thin driver (`classes12.zip`) is installed in the *`WL_HOME`*`\server\lib` folder (where *`WL_HOME`* is the folder where WebLogic Platform is installed) with `weblogic.jar`. The manifest in `weblogic.jar` lists this file so that it is loaded when `weblogic.jar` is loaded (when the server starts).

The *`WL_HOME`*`\server\ext\jdbc` folder (where *`WL_HOME`* is the folder where WebLogic Platform is installed) of your WebLogic Server installation includes subfolders for other versions of the Oracle Thin driver. See Figure 5-1.

Figure 5-1 Directory Structure for JDBC Drivers Installed with WebLogic Server



The `oracle` folder includes versions of the Oracle Thin driver, including the 9.2.0 version, which is also included in the *`WL_HOME`*`\server\lib` folder, as previously mentioned. You can copy one of these files to the *`WL_HOME`*`\server\lib` folder to change the version of the Oracle Thin driver or revert to the default version. See "Changing or Updating the Oracle Thin Driver" on page 5-4 for more details.

If you plan to use the default version of the Oracle Thin Driver (9.2.0), you do not need to make any changes. If you plan to use a different version of the driver, you must replace the file in *WL_HOME*\server\lib with a file from *WL_HOME*\server\ext\jdbc\oracle*version*, where *version* is the version of the JDBC driver you want to use, or with a file from Oracle.

Because the manifest in weblogic.jar lists the class files for the Oracle Thin driver in *WL_HOME*\server\lib, the drivers are loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add the JDBC driver to your CLASSPATH. If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you must add the path to the driver files to your CLASSPATH.

Setting the Environment for Your Third-Party JDBC Driver

If you use a third-party JDBC driver other than the Oracle Thin Driver included in the WebLogic Server installation, you must add the path for the JDBC driver classes to your CLASSPATH. The following sections describe how to set your CLASSPATH for Windows and UNIX when using a third-party JDBC driver.

CLASSPATH for Third-Party JDBC Driver on Windows

Include the path to JDBC driver classes and to weblogic.jar in your CLASSPATH as follows:

```
set CLASSPATH=DRIVER_CLASSES;WL_HOME\server\lib\weblogic.jar;
%CLASSPATH%
```

Where *DRIVER_CLASSES* is the path to the JDBC driver classes and *WL_HOME* is the directory where you installed WebLogic Platform.

CLASSPATH for Third-Party JDBC Driver on UNIX

Add the path to JDBC driver classes and to `weblogic.jar` to your `CLASSPATH` as follows:

```
export CLASSPATH=DRIVER_CLASSES:WL_HOME/server/lib/weblogic.jar:
$CLASSPATH
```

Where *`DRIVER_CLASSES`* is the path to the JDBC driver classes and *`WL_HOME`* is the directory where you installed WebLogic Platform.

Changing or Updating the Oracle Thin Driver

WebLogic Server ships with the Oracle Thin Driver version 9.2.0 preconfigured and ready to use. To use a different version, you replace *`WL_HOME`*`\server\lib\classes12.zip` with a different version of the file. For example, if you want to use the 8.1.7 version of the Oracle Thin Driver, you must copy `classes12.zip` from the *`WL_HOME`*`\server\ext\jdbc\oracle\817` folder and place it in *`WL_HOME`*`\server\lib` to replace the 9.2.0 version in that folder.

Follow these instructions to use Oracle Thin Driver version 8.1.7 or 9.0.1:

1. In Windows Explorer or a command shell, go to the folder for the version of the driver you want to use:
   - *`WL_HOME`*`\server\ext\jdbc\oracle\817` or
   - *`WL_HOME`*`\server\ext\jdbc\oracle\901`
2. Copy `classes12.zip`.
3. In Windows Explorer or a command shell, go to *`WL_HOME`*`\server\lib` and replace the existing version of `classes12.zip` with the version you copied.

To revert to version 9.2.0 (the default), follow the instructions above, but copy from the following folder: *`WL_HOME`*`\server\ext\jdbc\oracle\920`.

To update a version of the Oracle Thin driver with a new version from Oracle, replace `classes12.zip` in *`WL_HOME`*`\server\lib` with the new file from Oracle. You can download driver updates from the Oracle Web site at `http://otn.oracle.com/software/content.html`.

Note: You cannot include the multiple versions of the Oracle Thin driver in your `CLASSPATH`. Doing so will cause clashes for various methods.

Installing and Using the IBM Informix JDBC Driver

If you want to use Weblogic Server with an Informix database, BEA recommends that you use the IBM Informix JDBC driver, available from the IBM Web site at `http://www.informix.com/evaluate/`. The IBM Informix JDBC driver is available to use for free without support. You may have to register with IBM to download the product. Download the driver from the JDBC/EMBEDDED SQLJ section, and follow the instructions in the `install.txt` file included in the downloaded zip file to install the driver.

After you download and install the driver, follow these steps to prepare to use the driver with WebLogic Server:

1. Copy `ifxjdbc.jar` and `ifxjdbcx.jar` files from *`INFORMIX_INSTALL`*`\lib` and paste it in *`WL_HOME`*`\server\lib` folder, where:

   `INFORMIX_INSTALL` is the root directory where you installed the Informix JDBC driver, and

   `WL_HOME` is the folder where you installed WebLogic Platform, typically `c:\bea\weblogic700`.

2. Add the path to `ifxjdbc.jar` and `ifxjdbcx.jar` to your `CLASSPATH`. For example:

   ```
   set
   CLASSPATH=%WL_HOME%\server\lib\ifxjdbc.jar;%WL_HOME%\server\lib
   \ifxjdbcx.jar;%CLASSPATH%
   ```

   You can also add the path for the driver files to the `set CLASSPATH` statement in your start script for WebLogic Server.

Connection Pool Attributes when using the IBM Informix JDBC Driver

Use the attributes as described in Table 5-1 and Table 5-2 when creating a connection pool that uses the IBM Informix JDBC driver.

Table 5-1 Non-XA Connection Pool Attributes Using the Informix JDBC Driver

Attribute	Value
URL	`jdbc:informix-sqli:`*`dbserver_name_or_ip:port/dbname:`*`informixserver=`*`ifx_server_name`*
Driver Class Name	`com.informix.jdbc.IfxDriver`
Properties	`user=`*`username`* `url=jdbc:informix-sqli:`*`dbserver_name_or_ip:port/dbname:`*`informixserver=`*`ifx_server_name`* `portNumber=1543` `databaseName=`*`dbname`* `ifxIFXHOST=`*`ifx_server_name`* `serverName=`*`dbserver_name_or_ip`*
Password	*`password`*
Login Delay Seconds	`1`
Target	`serverName`

An entry in the `config.xml` file may look like the following:

```
<JDBCConnectionPool
    DriverName="com.informix.jdbc.IfxDriver"
    InitialCapacity="3"
    LoginDelaySeconds="1"
    MaxCapacity="10"
    Name="ifxPool"
    Password="xxxxxxx"
    Properties="informixserver=ifxserver;user=informix"
    Targets="examplesServer"
    URL="jdbc:informix-sqli:ifxserver:1543"
/>
```

Table 5-2 XA Connection Pool Attributes Using the Informix JDBC Driver

Attribute	Value
URL	*leave blank*
Driver Class Name	`com.informix.jdbcx.IfxXADataSource`
Properties	`user=`*`username`* `url=jdbc:informix-sqli://`*`dbserver_name_or_ip`*`:`*`port_num/dbname`*`:informixserver=`*`dbserver_name_or_ip`* `password=`*`password`* `portNumber =`*`port_num`*`;` `databaseName=`*`dbname`* `serverName=`*`dbserver_name`* `ifxIFXHOST=`*`dbserver_name_or_ip`*
Password	*leave blank*
Supports Local Transaction	`true`
Target	*serverName*

Note: In the Properties string, there is a space between `portNumber` and `=`.

An entry in the `config.xml` file may look like the following:

```
<JDBCConnectionPool CapacityIncrement="2"
    DriverName="com.informix.jdbcx.IfxXADataSource"
    InitialCapacity="2" MaxCapacity="10"
    Name="informixXAPool"
    Properties="user=informix;url=jdbc:informix-sqli:
    //111.11.11.11:1543/db1:informixserver=lcsol15;
    password=informix;portNumber =1543;databaseName=db1;
    serverName=dbserver1;ifxIFXHOST=111.11.11.11"
    SupportsLocalTransaction="true" Targets="examplesServer"
    TestConnectionsOnReserve="true" TestTableName="emp"/>
```

Note: If you create the connection pool using the Administration Console, you may need to stop and restart the server before the connection pool will deploy properly on the target server. This is a known issue.

Programming Notes for the IBM Informix JDBC Driver

Consider the following limitations when using the IBM Informix JDBC driver:

- Always call `resultset.close()` and `statement.close()` methods to indicate to the driver that you are done with the statement/resultset. Otherwise, your program may not release all its resources on the database server.
- Batch updates fail if you attempt to insert rows with TEXT or BYTE columns unless the `IFX_USEPUT` environment variable is set to `1`.
- If the Java program sets autocommit mode to true during a transaction, IBM Informix JDBC Driver commits the current transaction if the JDK is version 1.4 and later, otherwise the driver rolls back the current transaction before enabling autocommit.

Installing and Using the SQL Server 2000 Driver for JDBC from Microsoft

The Microsoft SQL Server 2000 Driver for JDBC is available for download to all licensed SQL Server 2000 customers at no charge. The driver is a Type 4 JDBC driver that supports a subset of the JDBC 2.0 Optional Package. When you install the Microsoft SQL Server 2000 Driver for JDBC, the supporting documentation is optionally installed with it. You should refer to that documentation for the most comprehensive information about the driver. Also, see the release manifest at `http://msdn.microsoft.com/MSDN-FILES/027/001/779/JDBCRTMReleaseManifest.htm` for known issues.

Installing the MS SQL Server JDBC Driver on a Windows System

Follow these instructions to install the SQL Server 2000 Driver for JDBC on a Windows server:

1. Download the Microsoft SQL Server 2000 Driver for JDBC (`setup.exe` file) from the Microsoft MSDN Web site at `http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/MSDN-FILES/027/001/779/msdncompositedoc.xml`. Save the file in a temporary directory on your local computer.
2. Run `setup.exe` from the temporary directory and follow the instructions on the screen.
3. Add the path to the following files to your `CLASSPATH`:
   - *`install_dir`*`/lib/msbase.jar`
   - *`install_dir`*`/lib/msutil.jar`
   - *`install_dir`*`/lib/mssqlserver.jar`

   Where *`install_dir`* is the folder in which you installed the driver. For example:

   ```
   set CLASSPATH=install_dir\lib\msbase.jar;
   install_dir\lib\msutil.jar;install_dir\lib\mssqlserver.jar;
   %CLASSPATH%
   ```

Installing the MS SQL Server JDBC Driver on a Unix System

Follow these instructions to install the SQL Server 2000 Driver for JDBC on a UNIX server:

1. Download the Microsoft SQL Server 2000 Driver for JDBC (`mssqlserver.tar` file) from the Microsoft MSDN Web site at `http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/MSDN-FILES/027/001/779/msdncompositedoc.xml`. Save the file in a temporary directory on your local computer.
2. Change to the temporary directory and untar the contents of the file using the following command:

   ```
   tar -xvf mssqlserver.tar
   ```
3. Execute the following command to run the installation script:

   ```
   install.ksh
   ```
4. Follow the instructions on the screen. When prompted to enter an installation directory, make sure you enter the full path to the directory.
5. Add the path to the following files to your `CLASSPATH`:

- *install_dir*/lib/msbase.jar
- *install_dir*/lib/msutil.jar
- *install_dir*/lib/mssqlserver.jar

Where *install_dir* is the folder in which you installed the driver. For example:

```
export CLASSPATH=install_dir/lib/msbase.jar:
install_dir/lib/msutil.jar:install_dir/lib/mssqlserver.jar:
$CLASSPATH
```

Connection Pool Attributes when using the Microsoft SQL Server Driver for JDBC

Use the following attributes when creating a connection pool that uses the Microsoft SQL Server Driver for JDBC:

- Driver Name: `com.microsoft.jdbc.sqlserver.SQLServerDriver`
- URL: `jdbc:microsoft:sqlserver://server_name:1433`
- Properties:

  `user=<myuserid>`

  `databaseName=<dbname>`
- Password: *mypassword*

An entry in the config.xml file may look like the following:

```
<JDBCConnectionPool
    Name="mssqlDriverTestPool"
    DriverName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    URL="jdbc:microsoft:sqlserver://lcdbnt4:1433"
    Properties="databasename=lcdbnt4;user=sa"
    Password="{3DES}vlsUYhxlJ/I="
    InitialCapacity="4"
    CapacityIncrement="2"
    MaxCapacity="10"
    Targets="examplesServer"
/>
```

Getting a Connection with Your Third-Party Driver

The following sections describe how to get a database connection using a third-party, Type 4 driver, such as the Oracle Thin Driver and Sybase jConnect Driver. BEA recommends you use connection pools, data sources, and a JNDI lookup to establish your connection.

Using Connection Pools with a Third-Party Driver

First, you create the connection pool and data source using the Administration Console, then establish a connection using a JNDI Lookup.

Creating the Connection Pool and DataSource

See "Configuring and Using Connection Pools" on page 2-2 and "Configuring and Using DataSources" on page 2-18 for instructions to create a JDBC connection pool and a JDBC DataSource.

Using a JNDI Lookup to Obtain the Connection

To access the driver using JNDI, obtain a Context from the JNDI tree by providing the URL of your server, and then use that context object to perform a lookup using the DataSource Name.

For example, to access a DataSource called "`myDataSource`" that is defined in the Administration Console:

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");
```

```
  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
   java.sql.Connection conn = ds.getConnection();

   // You can now use the conn object to create
   //  a Statement object to execute
   //  SQL statements and process result sets:

Statement stmt = conn.createStatement();
stmt.execute("select * from someTable");
ResultSet rs = stmt.getResultSet();

   // Do not forget to close the statement and connection objects
   // when you are finished:

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
    // a failure occurred
  }
  finally {
    try {ctx.close();}
    catch (Exception e) {
      // a failure occurred
    }
  }
```

(Where *hostname* is the name of the machine running your WebLogic Server and `port` is the port number where that machine is listening for connection requests.)

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see Programming WebLogic JNDI at http://e-docs.bea.com/wls/docs81b/jndi/index.html.

Notice that the JNDI lookup is wrapped in a `try/catch` block in order to catch a failed look up and also that the context is closed in a `finally` block.

Getting a Physical Connection from a Connection Pool

When you get a connection from a connection pool, WebLogic Server provides a logical connection rather than a physical connection so that WebLogic Server can manage the connection with the connection pool. This is necessary to enable

connection pool features and to maintain the quality of connections provided to applications. In some cases, you may want to use a physical connection, such as if you need to pass the connection to a method that checks the class name of the object for a particular class. WebLogic Server includes the `getVendorConnection()` method in the `weblogic.jdbc.extensions.WLConnection` interface that you can use to get the underlying physical connection from a logical connection. See the WebLogic Javadocs at `http://e-docs.bea.com/wls/docs81b/javadocs/weblogic/jdbc/extensions/WLConnection.html`.

Note: BEA strongly discourages using a physical connection instead of a logical connection from a connection pool. See "Limitations for Using a Physical Connection" on page 5-14.

When you use a physical connection, WebLogic Server does not provide any services or management for the connection, including error handling, XA support, and so forth.

Because there is no way for WebLogic Server to guarantee the quality of the connection or to effectively manage the connection after the physical connection is exposed, the connection is not returned to the connection pool after you are finished using it. Instead, the physical connection is closed.

When you get the underlying physical connection from a logical connection, WebLogic Server schedules a thread to open a new connection to replace exposed connection in the connection pool. The corresponding statement cache, if statement caching is enabled, is also created for the new connection.

Code Sample for Getting a Physical Connection

To get a physical database connection, you first get a connection from a connection pool as described in "Using a JNDI Lookup to Obtain the Connection" on page 5-11, then cast the connection as a WLConnection and call `getVendorConnection()`. For example:

```
//Import this additional package and any vendor packages
//you may need.
import weblogic.jdbc.extensions.*
.
.
.
try {
  ctx = new InitialContext(ht);
 // Look up the data source on the JNDI tree and request
```

```
// a connection.
 javax.sql.DataSource ds
   = (javax.sql.DataSource) ctx.lookup ("myDataSource");
java.sql.Connection conn = ds.getConnection();

// You can now cast the conn object to a WLConnection
// interface and then get the underlying physical connection.

java.sql.Connection vendorConn =
  ((WLConnection)conn)).getVendorConnection()

// You could also cast the vendorConn object to a vendor
// interface, such as:
// oracle.jdbc.OracleConnection vendorConn =
//  ((WLConnection)conn)).getVendorConnection()
```

You can now use the connection as necessary. When you are finished with the connection, close it with `connection.close()`.

Limitations for Using a Physical Connection

BEA strongly discourages using a physical connection instead of a logical connection from a connection pool. However, if you must use a physical connection, for example, to create a STRUCT, consider the following costs and limitations:

- The physical connection can only be used in server-side code.
- When you use a physical connection, you lose all of the connection management benefits that WebLogic Server offer, error handling, statement caching, and so forth.
- The physical connection may not be able to participate in global transactions.
- The connection is not reused. When you close the connection, it is not returned to the connection pool. Instead, the physical connection is closed and the connection pool creates a new connection to replace the one passed as a physical connection. Because the connection is not reused, there is a performance loss when using a physical connection because of the following:
  - The physical connection must be replaced with a new database connection in the connection pool, which uses resources on both the application server and the database server.
  - The statement cache for the original connection is closed and a new cache is opened for the new connection. Therefore, the performance gains from using the statement cache are lost.

Using Vendor Extensions to JDBC Interfaces

Some database vendors provide additional proprietary methods for working with data from a database that uses their DBMS. These methods extend the standard JDBC interfaces. In previous releases of Weblogic Server, only specific JDBC extensions for a few vendors were supported. The current release of WebLogic Server supports most extension methods exposed as a public interface in the vendor's JDBC driver.

If the driver vendor does not expose the methods you need in a public interface, you should send a request to the vendor to expose the methods in a public interface. WebLogic Server does provide support for extension methods in the Oracle Thin Driver for ARRAYs, STRUCTs, and REFs, even though the extension methods are not exposed in a public interface. See "Using Oracle Extensions with the Oracle Thin Driver" on page 5-18.

In general, WebLogic Server supports using vendor extensions in server-side code. To use vendor extensions in client-side code, the object type or data type must be serializable. Exceptions to this are the following object types:

- CLOB
- BLOB
- InputStream
- OutputStream

WebLogic Server supports using these object types in client-side code.

Note: There are interoperability limitations when using different versions of WebLogic Server clients and servers. See "Support for Vendor Extensions Between Versions of Weblogic Server Clients and Servers" on page 5-34.

To use the extension methods exposed in the JDBC driver, you must include these steps in your application code:

- Import the driver interfaces from the JDBC driver used to create connections in the connection pool.
- Get a connection from the connection pool.
- Cast the connection object as the vendor's connection interface.

- Use the vendor extensions as described in the vendor's documentation.

The following sections provide details in code examples. For information about specific extension methods for a particular JDBC driver, refer to the documentation from the JDBC driver vendor.

Sample Code for Accessing Vendor Extensions to JDBC Interfaces

The following code examples use extension methods available in the Oracle Thin driver to illustrate how to use vendor extensions to JDBC. You can adapt these examples to fit methods exposed in your JDBC driver.

Import Packages to Access Vendor Extensions

Import the interfaces from the JDBC driver used to create the connection in the connection pool. This example uses interfaces from the Oracle Thin Driver.

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import oracle.jdbc.*;

// Import driver interfaces. The driver must be the same driver
// used to create the database connection in the connection pool.
```

Get a Connection

Establish the database connection using JNDI, DataSource and connection pool objects. For information, see "Using a JNDI Lookup to Obtain the Connection" on page 5-11.

```
// Get a valid DataSource object for a connection pool.
// Here we assume that getDataSource() takes
// care of those details.
javax.sql.DataSource ds = getDataSource(args);

// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Cast the Connection as a Vendor Connection

Now that you have the connection, you can cast it as a vendor connection. This example uses the OracleConnection interface from the Oracle Thin Driver.

```
orConn = (oracle.jdbc.OracleConnection)conn;
// This replaces the deprecated process of casting the connection
// to a weblogic.jdbc.vendor.oracle.OracleConnection. For example:
// orConn = (weblogic.jdbc.vendor.oracle.OracleConnection)conn;
```

Use Vendor Extensions

The following code fragment shows how to use the Oracle Row Prefetch method available from the Oracle Thin driver.

```
// Cast to OracleConnection and retrieve the
// default row prefetch value for this connection.

int default_prefetch =

((oracle.jdbc.OracleConnection)conn).getDefaultRowPrefetch();
// This replaces the deprecated process of casting the connection
// to a weblogic.jdbc.vendor.oracle.OracleConnection. For example:
// ((weblogic.jdbc.vendor.oracle.OracleConnection)conn).
//       getDefaultRowPrefetch();

System.out.println("Default row prefetch
    is " + default_prefetch);

java.sql.Statement stmt = conn.createStatement();

// Cast to OracleStatement and set the row prefetch
// value for this statement. Note that this
// prefetch value applies to the connection between
// WebLogic Server and the database.

      ((oracle.jdbc.OracleStatement)stmt).setRowPrefetch(20);

   // This replaces the deprecated process of casting the
   // statement to a weblogic.jdbc.vendor.oracle.OracleStatement.
   // For example:
   // ((weblogic.jdbc.vendor.oracle.OracleStatement)stmt).
   //       setRowPrefetch(20);

      // Perform a normal sql query and process the results...
      String query = "select empno,ename from emp";
      java.sql.ResultSet rs = stmt.executeQuery(query);
```

```
    while(rs.next()) {
       java.math.BigDecimal empno = rs.getBigDecimal(1);
       String ename = rs.getString(2);
       System.out.println(empno + "\t" + ename);
    }

    rs.close();
    stmt.close();

    conn.close();
    conn = null;
  }
```

Using Oracle Extensions with the Oracle Thin Driver

For most extensions that Oracle provides, you can use the standard technique as described in "Using Vendor Extensions to JDBC Interfaces" on page 5-15. However, the Oracle Thin driver does not provide public interfaces for its extension methods in the following classes:

- `oracle.sql.ARRAY`
- `oracle.sql.STRUCT`
- `oracle.sql.REF`
- `oracle.sql.BLOB`
- `oracle.sql.CLOB`

WebLogic Server provides its own interfaces to access the extension methods for those classes:

- `weblogic.jdbc.vendor.oracle.OracleArray`
- `weblogic.jdbc.vendor.oracle.OracleStruct`
- `weblogic.jdbc.vendor.oracle.OracleRef`
- `weblogic.jdbc.vendor.oracle.OracleThinBlob`
- `weblogic.jdbc.vendor.oracle.OracleThinClob`

The following sections provide code samples for using the Weblogic Server interfaces for Oracle extensions. For a list of supported methods, see "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-35. For more information, please refer to the Oracle documentation.

Note: You can use this process to use any of the Weblogic Server interfaces for Oracle extensions listed in the "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-35. However, all but the interfaces listed above are deprecated and will be removed in a future release of WebLogic Server.

Limitations When Using Oracle JDBC Extensions

Please note the following limitations when using Oracle extensions to JDBC interfaces:

- You can use Oracle extensions for ARRAYs, REFs, and STRUCTs in server-side applications that use the same JVM as the server only. You cannot use Oracle extensions for ARRAYs, REFs, and STRUCTs in client applications.
- You cannot create ARRAYs, REFs, and STRUCTs in your applications. You can only retrieve existing ARRAY, REF, and STRUCT objects from a database. To create these objects in your applications, you must use a non-standard Oracle descriptor object, which is not supported in WebLogic Server.
- There are interoperability limitations when using different versions of WebLogic Server clients and servers. See "Support for Vendor Extensions Between Versions of Weblogic Server Clients and Servers" on page 5-34.

Sample Code for Accessing Oracle Extensions to JDBC Interfaces

The following code examples show how to access the WebLogic Server interfaces for Oracle extensions that are not available as public interfaces, including interfaces for:

- ARRAYs—See "Programming with ARRAYs" on page 5-20.
- STRUCTS—See "Programming with STRUCTs" on page 5-22.

- REFs—See "Programming with REFs" on page 5-27.
- BLOBs and CLOBs—See "Programming with BLOBs and CLOBs" on page 5-32.

If you selected the option to install server examples with WebLogic Server, see the JDBC examples for more code examples, typically at *WL_HOME*`\samples\server\src\examples\jdbc`, where *WL_HOME* is the folder where you installed WebLogic Server.

Programming with ARRAYs

In your WebLogic Server server-side applications, you can materialize an Oracle Collection (a SQL ARRAY) in a result set or from a callable statement as a Java array.

To use ARRAYs in WebLogic Server applications:

1. Import the required classes.
2. Get a connection and then create a statement for the connection.
3. Get the ARRAY using a result set or a callable statement.
4. Use the ARRAY as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
5. Use the standard Java methods (when used as a `java.sql.Array`) or Oracle extension methods (when cast as a `weblogic.jdbc.vendor.oracle.OracleArray`) to work with the data.

The following sections provide more details for these actions.

Note: You can use ARRAYs in server-side applications only. You cannot use ARRAYs in client applications.

Import Packages to Access Oracle Extensions

Import the Oracle interfaces used in this example. The OracleArray interface is counterpart to `oracle.sql.ARRAY` and can be used in the same way as the Oracle interface when using the methods supported by WebLogic Server.

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import weblogic.jdbc.vendor.oracle.*;
```

Establish the Connection

Establish the database connection using JNDI, DataSource and connection pool objects. For information, see "Using a JNDI Lookup to Obtain the Connection" on page 5-11.

```
// Get a valid DataSource object for a connection pool.
// Here we assume that getDataSource() takes
// care of those details.
javax.sql.DataSource ds = getDataSource(args);

// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Getting an ARRAY

You can use the `getArray()` methods for a callable statement or a result set to get a Java array. You can then use the array as a `java.sql.array` to use standard `java.sql.array` methods, or you can cast the array as a `weblogic.jdbc.vendor.oracle.OracleArray` to use the Oracle extension methods for an array.

The following example shows how to get a `java.sql.array` from a result set that contains an ARRAY. In the example, the query returns a result set that contains an object column—an ARRAY of test scores for a student.

```
try {

  conn = getConnection(url);
  stmt = conn.createStatement();
  String sql = "select * from students";
//Get the result set
  rs = stmt.executeQuery(sql);

  while(rs.next()) {
    BigDecimal id = rs.getBigDecimal("student_id");
    String name   = rs.getString("name");
   log("ArraysDAO.getStudents() -- Id = "+id.toString()+", Student
= "+name);
```

```
//Get the array from the result set
    Array scoreArray  = rs.getArray("test_scores");
    String[] scores = (String[])scoreArray.getArray();
    for (int i = 0; i < scores.length; i++) {
      log("   Test"+(i+1)+" = "+scores[i]);
    }
  }
```

Updating ARRAYs in the Database

To update an ARRAY in a database, you can use the `setArray()` method for a prepared statement or a callable statement. For example:

```
String sqlUpdate = "UPDATE SCOTT." + tableName + " SET col1 = ?";
conn  = ds.getConnection();
pstmt = conn.prepareStatement(sqlUpdate);
pstmt.setArray(1, array);
pstmt.executeUpdate();
```

Using Oracle Array Extension Methods

To use the Oracle extension methods for an ARRAY, you must first cast the array as a `weblogic.jdbc.vendor.oracle.OracleArray`. You can then make calls to the Oracle extension methods for ARRAYs. For example:

```
oracle.sql.Datum[] oracleArray = null;
oracleArray =
((weblogic.jdbc.vendor.oracle.OracleArray)scoreArray).getOracleArray();
String sqltype = null
sqltype = oracleArray.getSQLTypeName()
```

Programming with STRUCTs

In your WebLogic Server applications, you can access and manipulate *objects* from an Oracle database. When you retrieve objects from an Oracle database, you can cast them as either custom Java objects or as STRUCTs (`java.sql.struct` or `weblogic.jdbc.vendor.oracle.OracleStruct`). A STRUCT is a loosely typed data type for structured data which takes the place of custom classes in your applications. The STRUCT interface in the JDBC API includes several methods for

manipulating the attribute values in a STRUCT. Oracle extends the STRUCT interface with several additional methods. WebLogic Server implements all of the standard methods and most of the Oracle extensions.

Note: Please note the following limitations when using STRUCTs:

- STRUCTs are supported for use with Oracle only. To use STRUCTs in your applications, you must use the Oracle Thin Driver to communicate with the database, typically through a connection pool. The WebLogic jDriver for Oracle does not support the STRUCT data type.
- You can use STRUCTs in server-side applications only. You cannot use STRUCTs in client applications.

To use STRUCTs in WebLogic Server applications:

1. Import the required classes. (See "Import Packages to Access Oracle Extensions" on page 5-20.)
2. Get a connection. (See "Establish the Connection" on page 5-21.)
3. Use `getObject` to get the STRUCT.
4. Cast the STRUCT as a STRUCT, either `java.sql.Struct` (to use standard methods) or `weblogic.jdbc.vendor.oracle.OracleStruct` (to use standard and Oracle extension methods).
5. Use the standard or Oracle extension methods to work with the data.

The following sections provide more details for steps 3 through 5.

Getting a STRUCT

To get a database object as a STRUCT, you can use a query to create a result set and then use the `getObject` method to get the STRUCT from the result set. You then cast the STRUCT as a `java.sql.Struct` so you can use the standard Java methods. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs   = stmt.executeQuery("select * from people");
struct = (java.sql.Struct)(rs.getObject(2));
```

```
Object[] attrs = ((java.sql.Struct)struct).getAttributes();
```

WebLogic Server supports all of the JDBC API methods for STRUCTs:

- `getAttributes()`
- `getAttributes(java.util.Dictionary map)`
- `getSQLTypeName()`

Oracle supports the standard methods as well as the Oracle extensions. Therefore, when you cast a STRUCT as a `weblogic.jdbc.vendor.oracle.OracleStruct`, you can use both the standard and extension methods.

Using OracleStruct Extension Methods

To use the Oracle extension methods for a STRUCT, you must cast the `java.sql.Struct` (or the original `getObject` result) as a `weblogic.jdbc.vendor.oracle.OracleStruct`. For example:

```
java.sql.Struct struct =
(weblogic.jdbc.vendor.oracle.OracleStruct)(rs.getObject(2));
```

WebLogic Server supports the following Oracle extensions:

- `getDescriptor()`
- `getOracleAttributes()`
- getAutoBuffering()
- setAutoBuffering(boolean)

Getting STRUCT Attributes

To get the value for an individual attribute in a STRUCT, you can use the standard JDBC API methods `getAttributes()` and `getAttributes(java.util.Dictionary map)`, or you can use the Oracle extension method `getOracleAttributes()`.

To use the standard method, you can create a result set, get a STRUCT from the result set, and then use the `getAttributes()` method. The method returns an array of ordered attributes. You can assign the attributes from the STRUCT (object in the database) to an object in the application, including Java language types. You can then manipulate the attributes individually. For example:

```
conn = ds.getConnection();

stmt = conn.createStatement();

rs   = stmt.executeQuery("select * from people");

//The third column uses an object data type.
//Use getObject() to assign the object to an array of values.

struct = (java.sql.Struct)(rs.getObject(2));

Object[] attrs = ((java.sql.Struct)struct).getAttributes();

String address = attrs[1];
```

In the preceding example, the third column in the `people` table uses an object data type. The example shows how to assign the results from the `getObject` method to a Java object that contains an array of values, and then use individual values in the array as necessary.

You can also use the `getAttributes(java.util.Dictionary map)` method to get the attributes from a STRUCT. When you use this method, you must provide a hash table to map the data types in the Oracle object to Java language data types. For example:

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("NUMBER", Class.forName("java.lang.Integer"));

map.put("VARCHAR", Class.forName("java.lang.String"));

Object[] attrs = ((java.sql.Struct)struct).getAttributes(map);

String address = attrs[1];
```

You can also use the Oracle extension method `getOracleAttributes()` to get the attributes for a STRUCT. You must first cast the STRUCT as a `weblogic.jdbc.vendor.oracle.OracleStruct`. This method returns a datum array of `oracle.sql.Datum` objects. For example:

```
oracle.sql.Datum[] attrs =
 ((weblogic.jdbc.vendor.oracle.OracleStruct)struct).getOracleAttributes();

      oracle.sql.STRUCT address = (oracle.sql.STRUCT) attrs[1];

      Object address_attrs[] = address.getAttributes();
```

The preceding example includes a nested STRUCT. That is, the second attribute in the datum array returned is another STRUCT.

Using STRUCTs to Update Objects in the Database

To update an object in the database using a STRUCT, you can use the `setObject` method in a prepared statement. For example:

```
conn  = ds.getConnection();

stmt = conn.createStatement();

ps = conn.prepareStatement ("UPDATE SCHEMA.people SET EMPLNAME = ?,
 EMPID = ? where EMPID = 101");

ps.setString (1, “Smith”);

ps.setObject (2, struct);

ps.executeUpdate();
```

WebLogic Server supports all three versions of the `setObject` method.

Creating Objects in the Database

STRUCTs are typically used to materialize database objects in your Java application in place of custom Java classes that map to the database objects. In WebLogic Server applications, you cannot create STRUCTs that transfer to the database. However, you can use statements to create objects in the database that you can then retrieve and manipulate in your application. For example:

```
conn = ds.getConnection();

stmt = conn.createStatement();

cmd = "create type ob as object (ob1 int, ob2 int)"

stmt.execute(cmd);

cmd = "create table t1 of type ob";

stmt.execute(cmd);

cmd = "insert into t1 values (5, 5)"

stmt.execute(cmd);
```

Note: You cannot create STRUCTs in your applications. You can only retrieve existing objects from a database and cast them as STRUCTs. To create STRUCT objects in your applications, you must use a non-standard Oracle STRUCT descriptor object, which is not supported in WebLogic Server.

Automatic Buffering for STRUCT Attributes

To enhance the performance of your WebLogic Server applications that use STRUCTs, you can toggle automatic buffering with the `setAutoBuffering(boolean)` method. When automatic buffering is set to `true`, the `weblogic.jdbc.vendor.oracle.OracleStruct` object keeps a local copy of all the attributes in the STRUCT in their converted form (materialized from SQL to Java language objects). When your application accesses the STRUCT again, the system does not have to convert the data again.

Note: Buffering the converted attributes my cause your application to use an excessive amount of memory. Consider potential memory usage when deciding to enable or disable automatic buffering.

The following example shows how to activate automatic buffering:

```
((weblogic.jdbc.vendor.oracle.OracleStruct)struct).setAutoBuffering(true);
```

You can also use the `getAutoBuffering()` method to determine the automatic buffering mode.

Programming with REFs

A REF is a logical pointer to a row object. When you retrieve a REF, you are actually getting a pointer to a value in another table. The REF target must be a row in an object table. You can use a REF to examine or update the object it refers to. You can also change a REF so that it points to a different object of the same object type or assign it a null value.

Note: Please note the following limitations when using REFs:

- REFs are supported for use with Oracle only. To use REFs in your applications, you must use the Oracle Thin Driver to communicate with the database, typically through a connection pool. The WebLogic jDriver for Oracle does not support the REF data type.
- You can use REFs in server-side applications only.

To use REFs in WebLogic Server applications, follow these steps:

1. Import the required classes. (See "Import Packages to Access Oracle Extensions" on page 5-20.)
2. Get a database connection. (See "Establish the Connection" on page 5-21.)
3. Get the REF using a result set or a callable statement.
4. Cast the result as a STRUCT or as a Java object. You can then manipulate data using STRUCT methods or methods for the Java object.

You can also create and update a REF in the database.

The following sections describe these steps 3 and 4 in greater detail.

Getting a REF

To get a REF in an application, you can use a query to create a result set and then use the `getRef` method to get the REF from the result set. You then cast the REF as a `java.sql.Ref` so you can use the built-in Java method. For example:

```
conn = ds.getConnection();

stmt = conn.createStatement();

rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");

rs.next();

//Cast as a java.sql.Ref and get REF

ref = (java.sql.Ref) rs.getRef(1);
```

Note that the WHERE clause in the preceding example uses dot notation to specify the attribute in the referenced object.

After you cast the REF as a `java.sql.Ref`, you can use the Java API method `getBaseTypeName`, the only JDBC 2.0 standard method for REFs.

When you get a REF, you actually get a pointer to a value in an object table. To get or manipulate REF values, you must use the Oracle extensions, which are only available when you cast the `sql.java.Ref` as a `weblogic.jdbc.vendor.oracle.OracleRef`.

Using OracleRef Extension Methods

In order to use the Oracle extension methods for REFs, you must cast the REF as an Oracle REF. For example:

```
oracle.sql.StructDescriptor desc =
((weblogic.jdbc.vendor.oracle.OracleRef)ref).getDescriptor();
```

WebLogic Server supports the following Oracle extensions:

- `getDescriptor()`
- `getSTRUCT()`
- `getValue()`
- `getValue(dictionary)`
- `setValue(object)`

Getting a Value

Oracle provides two versions of the `getValue()` method—one that takes no parameters and one that requires a hash table for mapping return types. When you use either version of the `getValue()` method to get the value of an attribute in a REF, the method returns a either a STRUCT or a Java object.

The example below shows how to use the `getValue()` method without parameters. In this example, the REF is cast as an oracle.sql.STRUCT. You can then use the STRUCT methods to manipulate the value, as illustrated with the `getAttributes()` method.

```
oracle.sql.STRUCT student1 =
(oracle.sql.STRUCT)((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue ();

Object attributes[] = student1.getAttributes();
```

You can also use the `getValue(dictionary)` method to get the value for a REF. You must provide a hash table to map data types in each attribute of the REF to Java language data types. For example:

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("VARCHAR", Class.forName("java.lang.String"));

map.put("NUMBER", Class.forName("java.lang.Integer"));

oracle.sql.STRUCT result = (oracle.sql.STRUCT)
 ((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue (map);
```

Updating REF Values

When you update a REF, you can do any of the following:

- Change the value in the underlying table with the `setValue(object)` method.
- Change the location to which the REF points with a prepared statement or a callable statement.
- Set the value of the REF to null.

To use the `setValue(object)` method to update a REF value, you create an object with the new values for the REF, and then pass the object as a parameter of the `setValue` method. For example:

```
STUDENT s1 = new STUDENT();

s1.setName("Terry Green");

s1.setAge(20);

((weblogic.jdbc.vendor.oracle.OracleRef)ref).setValue(s1);
```

When you update the value for a REF with the `setValue(object)` method, you actually update the value in the table to which the REF points.

To update the *location* to which a REF points using a prepared statement, you can follow these basic steps:

1. Get a REF that points to the new location. You use this REF to replace the value of another REF.
2. Create a string for the SQL command to replace the location of an existing REF with the value of the new REF.
3. Create and execute a prepared statement.

For example:

```
try {

conn = ds.getConnection();

stmt = conn.createStatement();

//Get the REF.

rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
```

```
rs.next();

ref = (java.sql.Ref) rs.getRef(1); //cast the REF as a java.sql.Ref

}

//Create and execute the prepared statement.

String sqlUpdate = "update t3 s2 set col = ? where s2.col.ob1=20";

pstmt = conn.prepareStatement(sqlUpdate);

pstmt.setRef(1, ref);

pstmt.executeUpdate();
```

To use a callable statement to update the location to which a REF points, you prepare the stored procedure, set any IN parameters and register any OUT parameters, and then execute the statement. The stored procedure updates the REF value, which is actually a location. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");

rs.next();

ref1 = (java.sql.Ref) rs.getRef(1);

// Prepare the stored procedure

sql = "{call SP1 (?, ?)}";

cstmt = conn.prepareCall(sql);

// Set IN and register OUT params

cstmt.setRef(1, ref1);

cstmt.registerOutParameter(2, getRefType(), "USER.OB");

// Execute

cstmt.execute();
```

Creating a REF in the Database

You cannot create REF objects in your JDBC application—you can only retrieve existing REF objects from the database. However, you can create a REF in the database using statements or prepared statements. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
cmd = "create type ob as object (ob1 int, ob2 int)"
stmt.execute(cmd);
cmd = "create table t1 of type ob";
stmt.execute(cmd);
cmd = "insert into t1 values (5, 5)"
stmt.execute(cmd);
cmd = "create table t2 (col ref ob)";
stmt.execute(cmd);
cmd = "insert into t2 select ref(p) from t1 where p.ob1=5";
stmt.execute(cmd);
```

The preceding example creates an object type (`ob`), a table (`t1`) of that object type, a table (`t2`) with a REF column that can point to instances of `ob` objects, and inserts a REF into the REF column. The REF points to a row in `t1` where the value in the first column is `5`.

Programming with BLOBs and CLOBs

This section contains sample code that demonstrates how to access the OracleBlob interface. You can use the syntax of this example for the OracleBlob interface, when using methods supported by WebLogic Server. See "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-35.

Note: When working with BLOBs and CLOBs (referred to as "LOBs"), you must take transaction boundaries into account; for example, direct all read/writes to a particular LOB within a transaction. For additional information, refer to Oracle documentation about "LOB Locators and Transaction Boundaries" at the Oracle Web site at `http://www.oracle.com`.

Query to Select BLOB Locator from the DBMS

The BLOB Locator, or handle, is a reference to an Oracle Thin Driver BLOB:

```
String selectBlob = "select blobCol from myTable where blobKey =
666"
```

Declare the WebLogic Server java.sql Objects

The following code presumes the Connection is already established:

```
ResultSet rs = null;
Statement myStatement = null;
java.sql.Blob myRegularBlob = null;
java.io.OutputStream os = null;
```

Begin SQL Exception Block

In this try catch block, you get the BLOB locator and access the Oracle BLOB extension.

```
try {

     // get our BLOB locator..

     myStatement = myConnect.createStatement();
     rs = myStatement.executeQuery(selectBlob);
     while (rs.next()) {
       myRegularBlob = rs.getBlob("blobCol");

}

     // Access the underlying Oracle extension functionality for
     // writing. Cast to the OracleThinBlob interface to access
     // the Oracle method.

     os = ((OracleThinBlob)myRegularBlob).getBinaryOutputStream();
     .....
     ........

     } catch (SQLException sqe) {
       System.out.println("ERROR(general SQE): " +
         sqe.getMessage());
     }
```

Once you cast to the `Oracle.ThinBlob` interface, you can access the BEA supported methods.

Updating a CLOB Value Using a Prepared Statement

If you use a prepared statement to update a CLOB and the new value is shorter than the previous value, the CLOB will retain the characters that were not specifically replaced during the update. For example, if the current value of a CLOB is `abcdefghij` and you update the CLOB using a prepared statement with `zxyw`, the value in the CLOB is updated to `zxywefghij`. To correct values updated with a prepared statement, you should use the `dbms_lob.trim` procedure to remove the excess characters left after the update. See the Oracle documentation for more information about the `dbms_lob.trim` procedure.

Support for Vendor Extensions Between Versions of Weblogic Server Clients and Servers

Because the way WebLogic Server supports vendor JDBC extensions was changed in WebLogic Server 8.1, interoperability between versions of client and servers is affected.

When a WebLogic Server 8.1 client interacts with a WebLogic Server 7.0 or earlier server, Oracle extensions are not supported. When the client application tries to cast the JDBC objects to the Oracle extension interfaces, it will get a `ClassCastException`. However, when a WebLogic Server 7.0 or earlier client interacts with a WebLogic Server 8.1 server, Oracle extensions *are* supported.

This applies to the following Oracle extension interfaces:

- `weblogic.jdbc.vendor.oracle.OracleConnection`
- `weblogic.jdbc.vendor.oracle.OracleStatement`
- `weblogic.jdbc.vendor.oracle.OraclePreparedStatement`
- `weblogic.jdbc.vendor.oracle.OracleCallableStatement`
- `weblogic.jdbc.vendor.oracle.OracleResultSet.javaOracleThinBlob`
- `weblogic.jdbc.vendor.oracle.OracleThinClob`

- `weblogic.jdbc.vendor.oracle.OracleArray`
- `weblogic.jdbc.vendor.oracle.OracleRef`
- `weblogic.jdbc.vendor.oracle.OracleStruct`

Note: Standard JDBC interfaces are supported regardless of the client or server version.

Tables of Oracle Extension Interfaces and Supported Methods

In previous releases of Weblogic Server, only the JDBC extensions listed in the following tables were supported. The current release of WebLogic Server supports most extension methods exposed as a public interface in the vendor's JDBC driver. See "Using Vendor Extensions to JDBC Interfaces" on page 5-15 for instructions for using vendor extensions. Because the new internal mechanism for supporting vendor extensions does not rely on the previous implementation, several interfaces are no longer needed and are deprecated. These interfaces will be removed in a future release of Weblogic Server. See Table 5-3. BEA encourages you to use the alternative interface listed in the table.

Table 5-3 Deprecated Interfaces for Oracle JDBC Extensions

Deprecated Interface (supported in WebLogic Server 7.0 and earlier)	Instead, use this interface from Oracle (supported in WebLogic Server version 8.1 and later)
`weblogic.jdbc.vendor.oracle.OracleConnection`	`oracle.jdbc.OracleConnection`
`weblogic.jdbc.vendor.oracle.OracleStatement`	`oracle.jdbc.OracleStatement`
`weblogic.jdbc.vendor.oracle.OracleCallableStatement`	`oracle.jdbc.OracleCallableStatement`

Table 5-3 Deprecated Interfaces for Oracle JDBC Extensions

Deprecated Interface (supported in WebLogic Server 7.0 and earlier)	Instead, use this interface from Oracle (supported in WebLogic Server version 8.1 and later)
`weblogic.jdbc.vendor.oracle.OraclePreparedStatement`	`oracle.jdbc.OraclePreparedStatement`
`weblogic.jdbc.vendor.oracle.OracleResultSet`	`oracle.jdbc.OracleResultSet`

The interfaces listed in Table 5-4 are still valid because Oracle does not provide interfaces to access these extension methods.

Table 5-4 Oracle Interfaces with Continued Support in WebLogic Server

Oracle Interface
`weblogic.jdbc.vendor.oracle.OracleArray`
`weblogic.jdbc.vendor.oracle.OracleRef`
`weblogic.jdbc.vendor.oracle.OracleStruct`
`weblogic.jdbc.vendor.oracle.OracleThinClob`
`weblogic.jdbc.vendor.oracle.OracleThinBlob`

The following tables describe the Oracle interfaces and supported methods you use with the Oracle Thin Driver (or another driver that supports these methods) to extend the standard JDBC (`java.sql.*`) interfaces.

Table 5-5 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (This interface is **deprecated**. See Table 5-3.)	`boolean getAutoClose()`
	`  throws java.sql.SQLException;`
	
	`void setAutoClose(boolean on) throws`
	`  java.sql.SQLException;`
	
	`String getDatabaseProductVersion()`
	`  throws java.sql.SQLException;`
	
	`String getProtocolType() throws`
	`  java.sql.SQLException;`
	
	`String getURL() throws java.sql.SQLException;`
	
	`String getUserName()`
	`  throws java.sql.SQLException;`
	
	`boolean getBigEndian()`
	`  throws java.sql.SQLException;`
	
	`boolean getDefaultAutoRefetch() throws`
	`  java.sql.SQLException;`
	
	`boolean getIncludeSynonyms()`
	`  throws java.sql.SQLException;`
	
	`boolean getRemarksReporting()`
	`  throws java.sql.SQLException;`
	
	`boolean getReportRemarks()`
	`  throws java.sql.SQLException;`
	
	`boolean getRestrictGetTables()`
	`  throws java.sql.SQLException;`
	
	`boolean getUsingXAFlag()`
	`  throws java.sql.SQLException;`
	
	`boolean getXAErrorFlag()`
	`  throws java.sql.SQLException;`

Table 5-5 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-3.)	`boolean isCompatibleTo816()`
	`  throws java.sql.SQLException;`
	`  (`**`Deprecated`**`)`
	
	`byte[] getFDO(boolean b)`
	`  throws java.sql.SQLException;`
	
	`int getDefaultExecuteBatch() throws`
	`  java.sql.SQLException;`
	
	`int getDefaultRowPrefetch()`
	`  throws java.sql.SQLException;`
	
	`int getStmtCacheSize()`
	`  throws java.sql.SQLException;`
	
	`java.util.Properties getDBAccessProperties()`
	`  throws java.sql.SQLException;`
	
	`short getDbCsId() throws java.sql.SQLException;`
	
	`short getJdbcCsId() throws java.sql.SQLException;`
	
	`short getStructAttrCsId()`
	`  throws java.sql.SQLException;`
	
	`short getVersionNumber()`
	`  throws java.sql.SQLException;`
	
	`void archive(int i, int j, String s)`
	`  throws java.sql.SQLException;`
	
	`void close_statements()`
	`  throws java.sql.SQLException;`
	
	`void initUserName() throws java.sql.SQLException;`
	
	`void logicalClose() throws java.sql.SQLException;`
	
	`void needLine() throws java.sql.SQLException;`
	
	`void printState() throws java.sql.SQLException;`
	
	`void registerSQLType(String s, String t)`
	`  throws java.sql.SQLException;`
	
	`void releaseLine() throws java.sql.SQLException;`

Table 5-5 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued)
(This interface is **deprecated**. See Table 5-3.)
	`void removeAllDescriptor()`
	`  throws java.sql.SQLException;`
	
	`void removeDescriptor(String s)`
	`  throws java.sql.SQLException;`
	
	`void setDefaultAutoRefetch(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setDefaultExecuteBatch(int i)`
	`  throws java.sql.SQLException;`
	
	`void setDefaultRowPrefetch(int i)`
	`  throws java.sql.SQLException;`
	
	`void setFDO(byte[] b)`
	`  throws java.sql.SQLException;`
	
	`void setIncludeSynonyms(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setPhysicalStatus(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setRemarksReporting(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setRestrictGetTables(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setStmtCacheSize(int i)`
	`  throws java.sql.SQLException;`
	
	`void setStmtCacheSize(int i, boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setUsingXAFlag(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setXAErrorFlag(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void shutdown(int i)`
	`  throws java.sql.SQLException;`
	
	`void startup(String s, int i)`
	`  throws java.sql.SQLException;`

Table 5-6 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement` (This interface is **deprecated**. See Table 5-3.)	`String getOriginalSql()`
	`  throws java.sql.SQLException;`
	
	`String getRevisedSql()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated** in Oracle 8.1.7, removed in Oracle 9i.)`
	
	`boolean getAutoRefetch()`
	`  throws java.sql.SQLException;`
	
	`boolean is_value_null(boolean b, int i)`
	`  throws java.sql.SQLException;`
	
	`byte getSqlKind()`
	`  throws java.sql.SQLException;`
	
	`int creationState()`
	`  throws java.sql.SQLException;`
	
	`int getAutoRollback()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated**)`
	
	`int getRowPrefetch()`
	`  throws java.sql.SQLException;`
	
	`int getWaitOption()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated**)`
	
	`int sendBatch()`
	`  throws java.sql.SQLException;`
	
	`void clearDefines()`
	`  throws java.sql.SQLException;`
	
	`void defineColumnType(int i, int j)`
	`  throws java.sql.SQLException;`
	
	`void defineColumnType(int i, int j, String s)`
	`  throws java.sql.SQLException;`

Table 5-6 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement` (continued) (This interface is **deprecated**. See Table 5-3.)	`void defineColumnType(int i, int j, int k) throws java.sql.SQLException;`
	
	`void describe() throws java.sql.SQLException;`
	
	`void notify_close_rset() throws java.sql.SQLException;`
	
	`void setAutoRefetch(boolean b) throws java.sql.SQLException;`
	
	`void setAutoRollback(int i) throws java.sql.SQLException;` (**Deprecated**)
	
	`void setRowPrefetch(int i) throws java.sql.SQLException;`
	
	`void setWaitOption(int i) throws java.sql.SQLException;` (**Deprecated**)

Table 5-7 OracleResultSet Interface

Extends	Method Signature

`OracleResultSet` extends `java.sql.ResultSet` (This interface is **deprecated**. See Table 5-3.)	`boolean getAutoRefetch() throws java.sql.SQLException;`
	
	`int getFirstUserColumnIndex() throws java.sql.SQLException;`
	
	`void closeStatementOnClose() throws java.sql.SQLException;`
	
	`void setAutoRefetch(boolean b) throws java.sql.SQLException;`
	
	`java.sql.ResultSet getCursor(int n) throws java.sql.SQLException;`
	
	`java.sql.ResultSet getCURSOR(String s) throws java.sql.SQLException;`

.

Table 5-8 OracleCallableStatement Interface

Extends	Method Signature

`OracleCallableStatement` extends `java.sql.CallableStatement` (This interface is **deprecated**. See Table 5-3.)	`void clearParameters()`
	`  throws java.sql.SQLException;`
	
	`void registerIndexTableOutParameter(int i,`
	`  int j, int k, int l)`
	`  throws java.sql.SQLException;`
	
	`void registerOutParameter`
	`  (int i, int j, int k, int l)`
	`  throws java.sql.SQLException;`
	
	`java.sql.ResultSet getCursor(int i)`
	`  throws java.sql.SQLException;`
	
	`java.io.InputStream getAsciiStream(int i)`
	`  throws java.sql.SQLException;`
	
	`java.io.InputStream getBinaryStream(int i)`
	`  throws java.sql.SQLException;`
	
	`java.io.InputStream getUnicodeStream(int i)`
	`  throws java.sql.SQLException;`

.

Table 5-9 OraclePreparedStatement Interface

Extends	Method Signature

`OraclePreparedStatement` extends `OracleStatement` and `java.sql.PreparedStatement`
(This interface is **deprecated**. See Table 5-3.)
	`int getExecuteBatch()`
	`  throws java.sql.SQLException;`
	
	`void defineParameterType(int i, int j, int k)`
	`  throws java.sql.SQLException;`
	
	`void setDisableStmtCaching(boolean b)`
	`  throws java.sql.SQLException;`
	
	`void setExecuteBatch(int i)`
	`  throws java.sql.SQLException;`
	
	`void setFixedCHAR(int i, String s)`
	`  throws java.sql.SQLException;`
	
	`void setInternalBytes(int i, byte[] b, int j)`
	`  throws java.sql.SQLException;`

Table 5-10 OracleArray Interface

Extends	Method Signature

`OracleArray` extends `java.sql.Array`	`public ArrayDescriptor getDescriptor()`
	`  throws java.sql.SQLException;`
	
	`public Datum[] getOracleArray()`
	`  throws SQLException;`
	
	`public Datum[] getOracleArray(long l, int i)`
	`  throws SQLException;`
	
	`public String getSQLTypeName()`
	`  throws java.sql.SQLException;`
	
	`public int length()`
	`  throws java.sql.SQLException;`
	
	`public double[] getDoubleArray()`
	`  throws java.sql.SQLException;`
	
	`public double[] getDoubleArray(long l, int i)`
	`  throws java.sql.SQLException;`
	
	`public float[] getFloatArray()`
	`  throws java.sql.SQLException;`
	
	`public float[] getFloatArray(long l, int i)`
	`  throws java.sql.SQLException;`
	
	`public int[] getIntArray()`
	`  throws java.sql.SQLException;`
	
	`public int[] getIntArray(long l, int i)`
	`  throws java.sql.SQLException;`
	
	`public long[] getLongArray()`
	`  throws java.sql.SQLException;`
	
	`public long[] getLongArray(long l, int i)`
	`  throws java.sql.SQLException;`

Table 5-10 OracleArray Interface

Extends	Method Signature

`OracleArray` extends `java.sql.Array` (continued)	`public short[] getShortArray()`
	`  throws java.sql.SQLException;`
	
	`public short[] getShortArray(long l, int i)`
	`  throws java.sql.SQLException;`
	
	`public void setAutoBuffering(boolean flag)`
	`  throws java.sql.SQLException;`
	
	`public void setAutoIndexing(boolean flag)`
	`  throws java.sql.SQLException;`
	
	`public boolean getAutoBuffering()`
	`  throws java.sql.SQLException;`
	
	`public boolean getAutoIndexing()`
	`  throws java.sql.SQLException;`
	
	`public void setAutoIndexing(boolean flag, int i)`
	`  throws java.sql.SQLException;`

Table 5-11 OracleStruct Interface

Extends	Method Signature

`OracleStruct` extends `java.sql.Struct`	`public Object[] getAttributes()`
	`  throws java.sql.SQLException;`
	
	`public Object[] getAttributes(java.util.Dictionary map)`
	`  throws java.sql.SQLException;`
	
	`public Datum[] getOracleAttributes()`
	`  throws java.sql.SQLException;`
	
	`public oracle.sql.StructDescriptor getDescriptor()`
	`  throws java.sql.SQLException;`
	
	`public String getSQLTypeName()`
	`  throws java.sql.SQLException;`
	
	`public void setAutoBuffering(boolean flag)`
	`  throws java.sql.SQLException;`
	
	`public boolean getAutoBuffering()`
	`  throws java.sql.SQLException;`

Table 5-12 OracleRef Interface

Extends	Method Signature

`OracleRef` extends `java.sql.Ref`	`public String getBaseTypeName()`
	`  throws SQLException;`
	
	`public oracle.sql.StructDescriptor getDescriptor()`
	`  throws SQLException;`
	
	`public oracle.sql.STRUCT getSTRUCT()`
	`  throws SQLException;`
	
	`public Object getValue()`
	`  throws SQLException;`
	
	`public Object getValue(Map map)`
	`  throws SQLException;`
	
	`public void setValue(Object obj)`
	`  throws SQLException;`

Table 5-13 OracleThinBlob Interface

Extends	Method Signature

`OracleThinBlob` extends `java.sql.Blob`	`int getBufferSize()throws java.sql.Exception`
	
	`int getChunkSize()throws java.sql.Exception`
	
	`int putBytes(long, int, byte[])throws java.sql.Exception`
	
	`int getBinaryOutputStream()throws java.sql.Exception`

Table 5-14 OracleThinClob Interface

Extends	Method Signature

`OracleThinClob` extends `java.sql.Clob`	`public OutputStream getAsciiOutputStream() throws java.sql.Exception;`
	
	`public Writer getCharacterOutputStream() throws java.sql.Exception;`
	
	`public int getBufferSize() throws java.sql.Exception;`
	
	`public int getChunkSize() throws java.sql.Exception;`
	
	`public char[] getChars(long l, int i) throws java.sql.Exception;`
	
	`public int putChars(long start, char myChars[]) throws java.sql.Exception;`
	
	`public int putString(long l, String s) throws java.sql.Exception;`

CHAPTER

6 Testing JDBC Connections and Troubleshooting

The following sections describe how to test, monitor, and troubleshoot JDBC connections:

Monitoring JDBC Connectivity

The Administration Console provides tables and statistics to enable monitoring the connectivity parameters for each of the subcomponents—Connection Pools, MultiPools and DataSources.

You can also access statistics for connection pools programmatically through the `JDBCConnectionPoolRuntimeMBean`; see WebLogic Server Partner's Guide at `http://e-docs.bea.com/wls/docs81b/isv/index.html` and the WebLogic Javadoc. This MBean is the same API that populates the statistics in the Administration Console. Read more about monitoring connectivity in JDBC

Connection Pools at http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_connection_pools.html.

For information about using MBeans, see Programming WebLogic JMX Services at http://e-docs.bea.com/wls/docs81b/jmx/index.html.

Validating a DBMS Connection from the Command Line

Use BEA utilities to test two-tier and three-tier JDBC database connections after you install WebLogic Server.

Testing a Two-Tier Connection from the Command Line

To use the `utils.dbping` utility, you must complete the installation of your JDBC driver. Make sure you have completed the following:

- For Type 2 JDBC drivers, such as WebLogic jDriver for Oracle, set your `PATH` (Windows) or shared/load library path (UNIX) to include both your DBMS-supplied client installation and the BEA-supplied native libraries.
- For all drivers, include the classes of your JDBC driver in your `CLASSPATH`.
- Configuration instructions for the BEA WebLogic jDriver JDBC drivers are available at:
  - Using WebLogic jDriver for Oracle
  - Using WebLogic jDriver for Microsoft SQL Server

Use the `utils.dbping` utility to confirm that you can make a connection between Java and your database. The dbping utility is only for testing a two-tier connection, using a WebLogic two-tier JDBC driver like WebLogic jDriver for Oracle.

Syntax

```
$ java utils.dbping DBMS user password DB
```

Arguments

DBMS

Use: `ORACLE` or `MSSQLSERVER4`

user

Valid username for database login. Use the same values and format that you use with `isql` for SQL Server or `sqlplus` for Oracle.

password

Valid password for the user. Use the same values and format that you use with `isql` or `sqlplus`.

DB

Name of the database. The format varies depending on the database and version. Use the same values and format that you use with `isql` or `sqlplus`. Type 4 drivers, such as MSSQLServer4, need additional information to locate the server since they cannot access the environment.

Examples

Oracle

Connect to Oracle from Java with WebLogic jDriver for Oracle using the same values that you use with sqlplus.

If you are not using SQLNet (and you have `ORACLE_HOME and ORACLE_SID` defined), follow this example:

```
$ java utils.dbping ORACLE scott tiger
```

If you are using SQLNet V2, follow this example:

```
$ java utils.dbping ORACLE scott tiger TNS_alias
```

where `TNS_alias` is an alias defined in your local `tnsnames.ora` file.

Microsoft SQL Server (Type 4 driver)

To connect to Microsoft SQL Server from Java with WebLogic jDriver for Microsoft SQL Server, you use the same values for `user` and `password` that you use with `isql`. To specify the SQL Server, however, you supply the name of the computer running the SQL Server and the TCP/IP port the SQL Server is listening on. To log into a SQL Server running on a computer named `mars` listening on port 1433, enter:

```
$ java utils.dbping MSSQLSERVER4 sa secret mars:1433
```

You could omit "`:1433`" in this example since 1433 is the default port number for Microsoft SQL Server. By default, a Microsoft SQL Server may not be listening for TCP/IP connections. Your DBA can configure it to do so.

Validating a Multitier WebLogic JDBC Connection from the Command Line

Use the `utils.t3dbping` utility to confirm that you can make a multitier database connection using a WebLogic Server. The `t3dbping` utility is only for testing a multitier connection, after you have verified that you have a working two-tier connection, and after you have started WebLogic.

If the two-tier JDBC driver is a WebLogic jDriver, you should test the two-tier connection with `utils.dbping`. Otherwise, see the documentation for the two-tier JDBC driver to find out how to test that connection before you test the multitier connection.

Syntax

```
$ java utils.t3dbping URL user password DB driver_class driver_URL
```

Arguments

URL

URL of the WebLogic Server.

username

Valid username for the DBMS.

password

Valid password for that user.

DB

Name of the database. Use the same values and format that are shown above for testing a two-tier connection.

driver_class

Class name of the JDBC driver between WebLogic and the DBMS. For instance, if you are using WebLogic jDriver for Oracle on the server side, the driver class name is `weblogic.jdbc.oci.Driver`. Note that the class name of the driver is in dot-notation format.

driver_URL

URL of the JDBC driver between WebLogic and the DBMS. For instance, if you are using WebLogic jDriver for Oracle on the server side, the URL of the driver is `jdbc:weblogic:oracle`. Note that the URL of the driver is colon-separated.

Examples

These examples are displayed on multiple lines for readability. Each example should be entered as a single command.

Oracle

Here is an example of how to ping the Oracle DBMS DEMO20 running on the server bigbox, on the same host as WebLogic, which is listening on port 7001:

```
$ java utils.t3dbping          // command
    t3://bigbox:7001           // WebLogic URL
    scott tiger                // user password
     DEMO20                    // DB
     weblogic.jdbc.oci.Driver  // driver class
     jdbc:weblogic:oracle      // driver URL
```

DB2 with AS/400 Type 4 JDBC driver

This example shows how to ping an AS/400 DB2 database from a workstation command shell using the IBM AS/400 Type 4 JDBC driver:

```
$ java utils.t3dbping                       // command
     t3://as400box:7001                     // WebLogic URL
     scott tiger                            // user password
     DEMO                                   // database
     com.ibm.as400.access.AS400JDBCDriver   // driver class
     jdbc:as400://as400box                  // driver URL
```

WebLogic jDriver for Microsoft SQL Server (Type 4 JDBC driver)

This example shows how to ping a Microsoft SQL Server database using WebLogic jDriver for Microsoft SQL Server:

```
 $ java utils.t3dbping                  // command
      t3://localhost:7001               // WebLogic URL
      sa                                // user name
      abcd                              // password
```

```
    hostname                    // database@hostname:port
                                //(optional if specified
                                // as part of the URL)

weblogic.jdbc.mssqlserver4.Driver // driver class

jdbc:weblogic:mssqlserver4:pubs@localhost:1433

               // driver URL:database@hostname:port
               //(optional if used in the database parameter)
```

Troubleshooting JDBC

The following sections provide troubleshooting tips.

JDBC Connections

If you are testing a connection to WebLogic, check the WebLogic Server log. By default, the log is kept in a file with the following format:

domain`\`*server*`\`*server*`.log`

Where *domain* is the root folder of the domain and *server* is the name of the server. The server name is used as a folder name and in the log file name.

Windows

If you get an error message that indicates that the `.dll` failed to load, make sure your `PATH` includes the 32-bit database-related `.dlls`.

UNIX

If you get an error message that indicates that an `.so` or an `.sl` failed to load, make sure your `LD_LIBRARY_PATH` or `SHLIB_PATH` includes the 32-bit database-related files.

Codeset Support

WebLogic supports Oracle codesets with the following consideration:

- If your NLS_LANG environment variable is not set, or if it is set to either US7ASCII or WE8ISO8859-1, the driver always operates in 8859-1.
- If the NLS_LANG environment variable is set to a different value than the codeset used by the database, the Oracle Thin driver and the WebLogic jDriver for Oracle use the *client* codeset when writing to the database.

For more information, see Codeset Support in Using WebLogic jDriver for Oracle.

Other Problems with Oracle on UNIX

Check the threading model you are using. *Green* threads can conflict with the kernel threads used by OCI. When using Oracle drivers, WebLogic recommends that you use *native* threads. You can specify this by adding the `-native` flag when you start Java.

Thread-related Problems on UNIX

On UNIX, two threading models are available: green threads and native threads. For more information, read about the JDK for the Solaris operating environment on the Sun Web site at `http://www.java.sun.com`.

You can determine what type of threads you are using by checking the environment variable called `THREADS_TYPE`. If this variable is not set, you can check the shell script in your Java installation bin directory.

Some of the problems are related to the implementation of threads in the JVM for each operating system. Not all JVMs handle operating-system specific threading issues equally well. Here are some hints to avoid thread-related problems:

- If you are using Oracle drivers, use *native* threads.
- If you are using HP UNIX, upgrade to version 11.x, because there are compatibility issues with the JVM in earlier versions, such as HP UX 10.20.

- On HP UNIX, the new JDK does not append the green-threads library to the SHLIB_PATH. The current JDK can not find the shared library (.sl) unless the library is in the path defined by SHLIB_PATH. To check the current value of SHLIB_PATH, at the command line type:

  ```
  $ echo $SHLIB_PATH
  ```

  Use the set or setenv command (depending on your shell) to append the WebLogic shared library to the path defined by the symbol SHLIB_PATH. For the shared library to be recognized in a location that is not part of your SHLIB_PATH, you will need to contact your system administrator.

Closing JDBC Objects

BEA Systems recommends—and good programming practice dictates—that you always close JDBC objects, such as Connections, Statements, and ResultSets, in a finally block to make sure that your program executes efficiently. Here is a general example:

```
try {

Driver d =
(Driver)Class.forName("weblogic.jdbc.oci.Driver").newInstance();

Connection conn = d.connect("jdbc:weblogic:oracle:myserver",
                                  "scott", "tiger");

    Statement stmt = conn.createStatement();
    stmt.execute("select * from emp");
    ResultSet rs = stmt.getResultSet();
    // do work

    }

    catch (Exception e) {

      // deal with any exceptions appropriate

    }

    finally {

      try {rs.close();}
      catch (Exception rse) {}
      try {stmt.close();}
      catch (Exception sse) {}
```

```
        try {conn.close();
        catch (Exception cse) {}

    }
```

Troubleshooting Problems with Shared Libraries on UNIX

When you install a native two-tier JDBC driver, configure WebLogic Server to use performance packs, or set up BEA WebLogic Server as a Web server on UNIX, you install shared libraries or shared objects (distributed with the WebLogic Server software) on your system. This document describes problems you may encounter and suggests solutions for them.

The operating system loader looks for the libraries in different locations. How the loader works differs across the different flavors of UNIX. The following sections describe Solaris and HP-UX.

WebLogic jDriver for Oracle

Use the procedures for setting your shared libraries as described in this document. The actual path you specify will depend on your Oracle client version, your Oracle Server version and other factors. For details, see Installing WebLogic jDriver for Oracle.

Solaris

To find out which dynamic libraries are being used by an executable you can run the `ldd` command for the application. If the output of this command indicates that libraries are not found, then add the location of the libraries to the `LD_LIBRARY_PATH` environment variable as follows (for C or Bash shells):

```
# setenv LD_LIBRARY_PATH weblogic_directory/lib/solaris/oci817_8
```

Once you do this, `ld` should no longer complain about missing libraries.

HP-UX

Incorrectly Set File Permissions

The shared library problem you are most likely to encounter after installing WebLogic Server on an HP-UX system is incorrectly set file permissions. After installing WebLogic Server, make sure that the shared library permissions are set correctly with the `chmod` command. Here is an example to set the correct permissions for HP-UX 11.0:

```
% cd WL_HOME/lib/hpux11/oci817_8

% chmod 755 *.sl
```

If you encounter problems loading shared libraries *after* you set the file permissions, there could be a problem locating the libraries. First, make sure that the *WL_HOME*`/server/lib/hpux11` is in the `SHLIB_PATH` environment variable:

```
% echo $SHLIB_PATH
```

If the directory is not listed, add it:

```
# setenv SHLIB_PATH WL_HOME/server/lib/hpux11:$SHLIB_PATH
```

Alternatively, copy (or link) the .sl files from the WebLogic Server distribution to a directory that is already in the `SHLIB_PATH` variable.

If you still have problems, use the `chatr` command to specify that the application should search directories in the `SHLIB_PATH` environment variable. The `+s enabled` option sets an application to search the `SHLIB_PATH` variable. Here is an example of this command, run on the WebLogic jDriver for Oracle shared library for HP-UX 11.0:

```
# cd weblogic_directory/lib/hpux11

# chatr +s enable libweblogicoci38.sl
```

Check the `chatr` man page for more information on this command.

Incorrect SHLIB_PATH

You may also encounter a shared library problem if you do not include the proper paths in your `SHLIB_PATH` when using Oracle 9. SHLIB_PATH should include the path to the driver (`oci901_8`) and the path to the vendor-supplied libraries (`lib32`). For example, your path may look like:

```
export SHLIB_PATH=
$WL_HOME/server/lib/hpux11/oci901_8:$ORACLE_HOME/lib32:$SHLIB_PAT
H
```

Note also that your path cannot include the path to the Oracle 8.1.7 libraries, or clashes will occur. For more instructions, see Setting Up the Environment for Using WebLogic jDriver for Oracle at `http://e-docs.bea.com/wls/docs81b/oracle/install_jdbc.html`.