BEA WebLogic Server® and WebLogic Express®

Programming WebLogic JDBC

Version 8.1
Revised: June 28, 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

1. Introduction to WebLogic JDBC

2. Configuring and Using WebLogic JDBC

3. Performance Tuning Your JDBC Application

4. Using WebLogic Wrapper Drivers

5. Using Third-Party Drivers with WebLogic Server

6. Using RowSets with WebLogic Server

7. Testing JDBC Connections and Troubleshooting

A. Using WebLogic Server with Oracle RAC

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 Using 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 Wrapper 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, "Using RowSets with WebLogic Server," describes how to use WebLogic Server RowSets.
- Chapter 7, "Testing JDBC Connections and Troubleshooting," describes troubleshooting tips when using JDBC with WebLogic Server.
- Appendix A, "Using WebLogic Server with Oracle RAC," provides high level information for configuring WebLogic server when used with Oracle RAC to provide a more scalable and more available back-end system.

Audience

This document is written for application developers who want to build e-commerce applications using the Java 2 Platform, Enterprise Edition (J2EE). 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.

`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`*`;`

Convention	Usage
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Example*s: 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
. . .	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:

- *Standard JDBC drivers* that provide database access directly between a WebLogic Server 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.
- *Wrapper drivers* that provide vendor-neutral database access. A Java client application can use a wrapper driver to access any database configured in WebLogic server (via a connection pool). BEA offers three wrapper 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. A client application can also use these drivers directly to get a connection from a connection pool (You can use RMI from external clients and the pool and JTS from server-side clients only). However, BEA recommends that you use a data source to get a connection from a connection pool, rather than using these drivers directly. (See "Obtaining a Client Connection Using a DataSource" on page 2-15.)

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 wrapper 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

Type and Name of Driver	Database Connectivity	Documentation Sources
Type 2 (requires native libraries): • WebLogic jDriver for Oracle • WebLogic jDriver for Oracle XA • Third-party drivers, such as the Oracle OCI driver and the IBM DB2 driver	Between WebLogic Server and DBMS in local and distributed transactions.	*Programming WebLogic JDBC* (this document) *Programming WebLogic JTA* *Administration Console Online Help*, "Configuring JDBC Connection Pools" *Using WebLogic jDriver for Oracle*
Type 4 (pure Java) • WebLogic jDrivers for Microsoft SQL Server • Third-party drivers, including: Oracle Thin and Oracle Thin XA drivers	Between WebLogic Server and DBMS in local and distributed transactions. **Note:** The WebLogic jDrivers for Microsoft SQL Server supports local transactions only.	*Programming WebLogic JDBC* (this document) *Programming WebLogic JTA* *Administration Console Online Help*, "Configuring JDBC Connection Pools" *Using WebLogic jDriver for Microsoft SQL Server*
Type 3 • WebLogic RMI Driver	Between an external client and WebLogic Server (connection pool).	*Programming WebLogic JDBC* (this document)

Selecting a JDBC Driver

When deciding which JDBC driver to use to connect to a database, you should try drivers from various vendors in your environment. In general, JDBC driver performance is dependent on many factors, especially the SQL code used in applications and the JDBC driver implementation.

For information about supported JDBC drivers, see "Supported Database Configurations" in *Supported Configurations for WebLogic Platform 8.1*.

WebLogic Server JDBC Drivers

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

WebLogic jDriver for Oracle (Deprecated)

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/docs81oracle/index.html`.

BEA WebLogic Type 4 JDBC Driver for Microsoft SQL Servers

WebLogic Server 8.1 SP1 includes a new JDBC driver from BEA for connecting to a Microsoft SQL Server database. The BEA WebLogic Type 4 JDBC MS SQL Server driver replaces the WebLogic jDriver for Microsoft SQL Server, which is deprecated. The new driver offers JDBC 3.0 compliance, support for some JDBC 2.0 extensions, and better performance. BEA recommends that you use the new BEA WebLogic Type 4 JDBC MS SQL Server driver in place of the WebLogic jDriver for Microsoft SQL Server.

For more information, see *BEA WebLogic Type 4 JDBC Drivers*.

WebLogic jDriver for Microsoft SQL Server (Deprecated)

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/docs81/mssqlserver4/index.html`.

WebLogic Server Wrapper Drivers

The following sections briefly describe the WebLogic wrapper 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 Wrapper Drivers."

WebLogic RMI Driver

The WebLogic RMI driver is a Type 3 JDBC driver that runs in WebLogic Server. You can use the WebLogic RMI driver in a remote client application to connect to a database through a WebLogic Server 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. For requests from external clients, the data source then internally uses the RMI driver, if necessary.

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 wrapper 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 calls from the transaction manager to start and end work with this branch of the transaction (`XAResource.start()` and `XAResource.end()`) do not require communication with the database (they are no-ops).

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 JDBC 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.

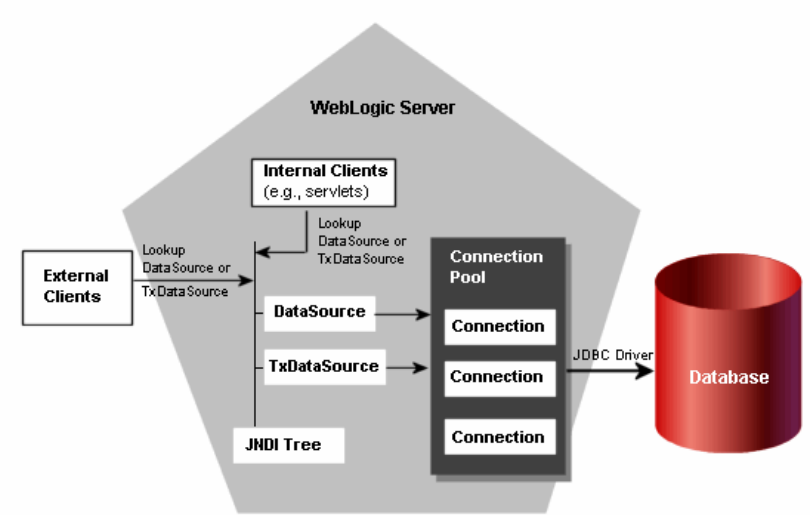
Oracle Thin Driver

The *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 WebLogic wrapper 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 JDBC driver to make the physical database connections from WebLogic Server to the DBMS. The JDBC driver can be one of the WebLogic jDrivers or a third-party JDBC driver, such as 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	Creating a DBMS connection is a slow operation. 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 distributed transactions, use a TxDataSource from the JNDI tree. For transactions distributed across multiple servers within a single WebLogic domain with one database instance, use a TxDataSource from the JNDI tree or use the JTS driver. Note that BEA recommends that you

access connection pools using the JNDI tree and a DataSource object rather than using WebLogic wrapper drivers.

Using Connection Pools with Client-side Applications

Note: For new deployments, BEA recommends that you use a DataSource from the JNDI tree to access database connections rather than the RMI driver.

BEA offers the RMI driver for client-side JDBC. The RMI driver provides a standards-based approach using the Java 2 Enterprise Edition (J2EE) specifications.

The WebLogic RMI driver is a Type 3 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

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, based on the selected algorithm.

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 2-35.

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/docs81/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 2-13.

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. See "Configuring and Using Connection Pools" on page 2-2. Also see the following WebLogic Javadocs:

- `weblogic.management.configuration` (MBeans for creating DataSources, connection pools, and MultiPools)
- `weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean` (MBean for runtime operations on a connection pool)

JDBC 2.0

WebLogic Server uses a Java 2 SDK 1.4.1, which supports JDBC 2.0.

Platforms

Supported platforms vary by vendor-specific DBMSs and drivers. For current information, see BEA WebLogic Supported Configurations at `http://e-docs.bea.com/platform/suppconfigs/index.html`.

CHAPTER 2

Configuring and Using WebLogic JDBC

You use the WebLogic Server Administration Console to enable, configure, and monitor features of WebLogic Server, including JDBC connection pools, data sources, and MultiPools. You can do the same tasks programmatically using the JMX API and the `weblogic.Admin` command line utility. After configuring JDBC connectivity components, you can use them in your applications.

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/docs81/ConsoleHelp/jdbc.html`.
- WebLogic Server Javadocs at `http://e-docs.bea.com/wls/docs81/javadocs/index.html` for the following interfaces and packages:
  - `weblogic.management.configuration.JDBCConnectionPoolMBean`
  - `weblogic.management.configuration.JDBCDataSourceFactoryMBean`

- `weblogic.management.configuration.JDBCDataSourceMBean`
- `weblogic.management.configuration.JDBCMultiPoolMBean`
- `weblogic.management.configuration.JDBCTxDataSourceMBean`
- `weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean`
- `weblogic.jdbc.extensions`

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 deployed, 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 username and 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 2-9. You can also use the command line; see the Web Logic Server Command-Line Interface Reference at `http://e-docs.bea.com/wls/docs81/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. 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/docs81/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. 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. BEA recommends that you use the Password attribute in place of the password property in the properties string because 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+Threads=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+Threads=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+Sqln
et=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.

SQL Statement Timeout Enhancements for Pooled JDBC Connections

In WebLogic Server 8.1SP3, the following attributes were added to JDBC connection pools to enable you to limit the amount of time that a statement can execute on a database connection from a JDBC connection pool:

- `StatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection times out. When set to -1, (the default) statements do not timeout.
- `TestStatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection for connection initialization or testing times out. When set to -1, (the default) statements do not timeout. See "Initializing Connections with a SQL Query" on page 2-20 and "Testing Connection Pools and Database Connections" on page 2-21 for more information about SQL statements used for initializing and testing connections.

These attributes rely on underlying JDBC driver support. WebLogic Server passes the time specified to the JDBC driver using the `java.sql.Statement.setQueryTimeout()` method. If your JDBC driver does not support this method, it may throw an exception and the timeout value is ignored.

Note: Using these features may cause a performance degradation. You should test these features in a staging or testing environment before using them in production.

Also, these attributes are not available in the Administration Console. You must manually edit the `config.xml` file to enable these features.

JDBC Connection Pool Testing Enhancements

In WebLogic Server 8.1SP3, the following attributes were added to JDBC connection pools to improve the functionality of database connection testing for pooled connections:

- `CountOfTestFailuresTillFlush`—Closes all connections in the connection pool after the number of test failures that you specify to minimize the delay caused by further database testing. See "Minimizing Connection Test Delay After Database Connectivity Loss."
- `CountOfRefreshFailuresTillDisable`—Disables the connection pool after the number of test failures that you specify to minimize the delay in handling the connection request after a database failure. See "Minimizing Connection Request Delay After Connection Test Failures."

- SecondsToTrustAnIdlePoolConnection—Skips the connection test if the connection was used or tested successfully within the time specified. See "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection" on page 2-8.

Minimizing Connection Test Delay After Database Connectivity Loss

When connectivity to the DBMS is lost, even if only momentarily, some or all of the JDBC connections in the connection pool typically become defunct. If the connection pool is configured to test connections on reserve (recommended), when an application requests a database connection, WebLogic Server tests the connection, discovers that the connection is dead, and tries to replace it with a new connection to satisfy the request. Ordinarily, when the DBMS comes back online, the refresh process succeeds. However, in some cases and for some modes of failure, testing a dead connection can impose a long delay. This delay occurs for each dead connection in the connection pool until all connections are replaced.

To minimize the delay that occurs during the test of dead database connections, you can set the `CountOfTestFailuresTillFlush` attribute on the connection pool. With this attribute set, WebLogic Server considers *all* connections in the connection pool as dead after the number of consecutive test failures that you specify, and closes all connections in the connection pool.

When an application requests a connection, the connection pool creates a connection without first having to test a dead connection. This behavior minimizes the delay for connection requests following the connection pool flush.

You specify the `CountOfTestFailuresTillFlush` attribute in the JDBCConnectionPool entry in the `config.xml` file. `TestConnectionsOnReserve` must also be set to `true`. For example:

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  TestConnectionsOnReserve="true"
  CountOfTestFailuresTillFlush="1"
  TestTableName="SYSTABLES"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

Note: The `CountOfTestFailuresTillFlush` attribute is not available in the Administration Console.

If you tend to see small network glitches or have a firewall that may occasionally kill only one socket or connection, you may want to set the number of test failures to 2 or 3, but a value of 1 will provide the best performance after database availability issues have been resolved.

Minimizing Connection Request Delay After Connection Test Failures

If your DBMS becomes and remains unavailable, the connection pool will persistently test and try to replace dead connections while trying to satisfy connection requests. This behavior is beneficial because it enables the connection pool to react immediately when the database becomes available. However, testing a dead database connection can take as long as the network timeout, and can cause a long delay for clients.

To minimize the delay that occurs for client applications while a database is unavailable, you can set the `CountOfRefreshFailuresTillDisable` attribute on the connection pool. With this attribute set, WebLogic Server disables the connection pool after the number of consecutive failures to replace a dead connection. When an application requests a connection from a disabled connection pool, WebLogic Server throws a `ConnectDisabledException` immediately to notify the client that a connection is not available.

For connection pools that are disabled in this manner, WebLogic Server periodically run the refresh process. When the refresh process succeeds in creating a new database connection, WebLogic Server re-enables the connection pool. You can also manually re-enable the connection pool using the administration console or the `weblogic.Admin ENABLE_POOL` command.

You specify the `CountOfRefreshFailuresTillDisable` attribute in the JDBCConnectionPool entry in the `config.xml` file. `TestConnectionsOnReserve` must also be set to `true`. For example:

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  TestConnectionsOnReserve="true"
  CountOfRefreshFailuresTillDisable="1"
```

```
  TestTableName="SYSTABLES"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

Note: The `CountOfRefreshFailuresTillDisable` attribute is not available in the Administration Console.

If you tend to see small network glitches or have a firewall that may occasionally kill only one socket or connection, you may want to set the number of refresh failures to 2 or 3, but a value of 1 will usually provide the best performance.

Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection

Database connection testing during heavy traffic can reduce application performance. To minimize the impact of connection testing, you can set the SecondsToTrustAnIdlePoolConnection attribute in the JDBC connection pool configuration to trust recently-used or recently-tested database connections as viable and skip the connection test.

If your connection pool is configured to test connections on reserve (recommended), when an application requests a database connection, WebLogic Server tests the database connection before giving it to the application. If the request is made within the time specified for SecondsToTrustAnIdlePoolConnection since the connection was tested or successfully used and returned to the connection pool, WebLogic Server skips the connection test before delivering it to the application.

If your connection pool is configured to periodically test available connections in the connection pool (TestFrequencySeconds is specified), WebLogic Server also skips the connection test if the connection was successfully used and returned to the connection pool within the time specified for SecondsToTrustAnIdlePoolConnection.

To set SecondsToTrustAnIdlePoolConnection, you can specify the value on the JDBC Connection Pool →Configuration →Connections tab in the Administration Console. See "JDBC Connection Pool --> Configuration --> Connections" in the *Administration Console Online Help*. You can also set it directly in the `config.xml` file. For example:

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
```

```
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  SecondsToTrustAnIdlePoolConnection="15"
  TestConnectionsOnreserve="true"
  TestTableName="SYSTABLES"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

SecondsToTrustAnIdlePoolConnection is a tuning feature that can improve application performance by minimizing the delay caused by database connection testing, especially during heavy traffic. However, it can reduce the effectiveness of connection testing, especially if the value is set too high. The appropriate value depends on your environment and the likelihood that a connection will become defunct.

Creating a Connection Pool Dynamically

The `JDBCConnectionPool` administration MBean as part of the WebLogic Server management architecture (JMX). You can use the JMX API 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 instance 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/docs81/admin_ref/cli.html#cli_create_pool`.

To dynamically create a connection pool using the JMX API, 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 JMX API, 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.

For more information about using MBeans to manage WebLogic Server, see Programming WebLogic Management Services with JMX at `http://e-docs.bea.com/wls/docs81/jmx/index.html`. For more information about the `JDBCConnectionPool` MBean, see the Javadoc at `http://e-docs.bea.com/wls/docs81/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.JDBCTxDataSourceMBean;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicObjectName;

String cpName = null;
String cpJNDIName = null;
```

Look Up the Administration MBeanHome

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

Get the Server MBean

```
svrAdminMBean = (ServerMBean)mbeanHome.getAdminMBean("myserver",
 "Server");
```

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", "dbserver1t1");

  // Set DataSource attributes
  cpMBean.setURL("jdbc:weblogic:oracle");
  cpMBean.setDriverName("weblogic.jdbc.oci.xa.XADataSource");
  cpMBean.setProperties(pros);
  cpMBean.setPassword("tiger");
```

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

When you add a deployment target, the connection pool is deployed and database connections in the connection pool are created.

```
  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 TxDataSource  MBean
      dsMBean = (JDBCTxDataSourceMBean)mbeanHome.createAdminMBean(
        cpName, "JDBCTxDataSource",
        mbeanHome.getDomainName());

      // Set TxDataSource attributes
      dsMBean.setJNDIName(cpJNDIName);
      dsMBean.setPoolName(cpName);

      // Startup datasource
      dsMBean.addTarget(serverMBean);

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

    }

  }
```

Note: The `JDBCDataSourceMBean` is deprecated in WebLogic server 8.1. Use the `JDBCTxDataSourceMBean` instead. The attributes that are not available in the `JDBCTxDataSourceMBean` (`WaitForConnectionEnabled` and `ConnectionWaitPeriod`) have been deprecated and are replaced with the `ConnectionReserveTimeoutSeconds` attribute in the `JDBCConnectionPoolMBean`. See "Enabling Connection Requests to Wait for a Connection" on page 2-22.

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 TxDataSource from the server

      dsMBean.removeTarget(serverMBean);

      // Remove dynamically created TxDataSource from the configuration

      mbeanHome.deleteMBean(dsMBean);

    } catch (Exception ex) {

      throw new SQLException(ex.toString());

    }

  }
```

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)

Note: In the Administration Console, Data Sources and Tx Data Sources are distinguished by the Honor Global Transactions setting that you select when you create the datasource:

`true` for Tx Data Sources

`false` for Data Sources (non-Tx)

Tx Data Sources are created by default when you create the data source in the Administration Console.

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 updates to multiple databases during a single transaction.

The only time you should use a non-Tx Data Source is when you want to do some work on the database that you do not want to include in the current transaction.

If you want applications to use a DataSource (Tx or non-Tx) 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/docs81/ConsoleHelp/jdbc_datasources.html`.

Note: The `JDBCDataSourceMBean` is deprecated in WebLogic server 8.1. Use the `JDBCTxDataSourceMBean` instead. The attributes that are not available in the `JDBCTxDataSourceMBean` (`WaitForConnectionEnabled` and `ConnectionWaitPeriod`) have been deprecated and are replaced with the `ConnectionReserveTimeoutSeconds` attribute in the `JDBCConnectionPoolMBean`. See "Enabling Connection Requests to Wait for a Connection" on page 2-22.

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 for a JDBC client, use a Java Naming and Directory Interface (JDNI) lookup to locate the DataSource object, as shown in the following code fragment.

Note: When using a JDBC connection in a client-side application, the *exact same* JDBC driver classes must be in the `CLASSPATH` on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

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

  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;

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

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

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

...

//Close JDBC objects as soon as possible
    stmt.close();
    stmt=null;

    conn.close();
    conn=null;

 }
  catch (Exception e) {
```

```
      // a failure occurred
      log message;
    }
  finally {
    try {
      ctx.close();
    } catch (Exception e) {
       log message; }
    try {
      if (rs != null) rs.close();
    } catch (Exception e) {
       log message; }
    try {
      if (stmt != null) stmt.close();
    } catch (Exception e) {
       log message; }
    try {
      if (conn != null) conn.close();
    } catch (Exception e) {
       log message; }
  }
```

(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/docs81/jndi/index.html`.

Possible Exceptions When a Connection Request Fails

The weblogic.jdbc.extensions package includes the following exceptions that can be thrown when an application request fails. Each exception extends `java.sql.SQLException`.

- `ConnectionDeadSQLException`—generated when an application request to get a connection fails because the connection test on the reserved connection failed. This typically happens when the database server is unavailable. See "Testing Connection Pools and Database Connections" on page 2-21.
- `ConnectionUnavailableSQLException`—generated when an application request to get a connection fails because there are currently no connections available in the pool to be allocated. This is a transient failure, and is generated if all connections in the pool are

currently in use. It can also be thrown when connections are unavailable because they are being tested. See "Testing Connection Pools and Database Connections" on page 2-21.

- `PoolDisabledSQLException`—generated when an application request to get a connection fails because the JDBC Connection Pool has been administratively disabled. See "Suspending a Connection Pool" on page 2-26.
- `PoolLimitSQLException`—generated when an application request to get a connection fails due to a configured threshold of the connection pool, such as `HighestNumWaiters`, `ConnectionReserveTimeoutSeconds`, and so forth. See "Enabling Connection Requests to Wait for a Connection" on page 2-22.
- `PoolPermissionsSQLException`—generated when an application request to get a connection fails a (security) authentication or authorization check.

Connection Pool Limitation

When using connection pools, it is possible to execute DBMS-specific SQL code that will alter the database connection properties and that WebLogic Server and the JDBC driver will be unaware of. When the connection is returned to the connection pool, the characteristics of the connection may not be set back to a valid state. For example, with a Sybase DBMS, if you use a statement such as `"set rowcount 3 select * from y"`, the connection will only ever return a maximum of 3 rows from any subsequent query on this connection. When the connection is returned to the connection pool and then reused, the next user of the connection will still only get 3 rows returned, even if the table being selected from has 500 rows.

In most cases, there is standard JDBC code that can accomplish the same result. In this example, you could use `setMaxRows()` instead of `set rowcount`. BEA recommends that you use the standard JDBC code instead of the DBMS-specific SQL code. When you use standard JDBC calls to alter the connection, Weblogic Server returns the connection to a standard state when the connection is returned to the connection pool.

If you use DBMS-specific SQL code that alters the connection, you must set the connection back to an acceptable state before returning the connection to the connection pool.

Managing Connection Pools

The `JDBCConnectionPool` and `JDBCConnectionPoolRuntime` MBeans provide methods to manage connection pools and obtain information about them. All of these management options are available in the Administration Console. However, you can also use the methods provided to

manage connection pools using the JMX API. Methods are provided for these and other operations:

To see all of the methods available and for more information about the methods described in this section, see the Javadocs for the following MBeans:

- JDBCConnectionPoolMBean at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html`
- JDBCConnectionPoolRuntimeMBean at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/management/runtime/JDBCConnectionPoolRuntimeMBean.html`

Getting Status and Statistics for a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.getState()

JDBCConnectionPoolRuntimeMBean.getActiveConnectionsAverageCount()
JDBCConnectionPoolRuntimeMBean.getActiveConnectionsCurrentCount()
JDBCConnectionPoolRuntimeMBean.getActiveConnectionsHighCount()
JDBCConnectionPoolRuntimeMBean.getConnectionLeakProfileCount()
JDBCConnectionPoolRuntimeMBean.getConnectionsTotalCount()
JDBCConnectionPoolRuntimeMBean.getCurrCapacity()
JDBCConnectionPoolRuntimeMBean.getFailuresToReconnectCount()
JDBCConnectionPoolRuntimeMBean.getHighestNumAvailable()
```

```
JDBCConnectionPoolRuntimeMBean.getHighestNumUnavailable()
JDBCConnectionPoolRuntimeMBean.getLeakedConnectionCount()
JDBCConnectionPoolRuntimeMBean.getMaxCapacity()
JDBCConnectionPoolRuntimeMBean.getNumAvailable()
JDBCConnectionPoolRuntimeMBean.getNumUnavailable()
JDBCConnectionPoolRuntimeMBean.getStatementProfileCount()
JDBCConnectionPoolRuntimeMBean.getVersionJDBCDriver()
JDBCConnectionPoolRuntimeMBean.getWaitingForConnectionCurrentCount()
JDBCConnectionPoolRuntimeMBean.getWaitingForConnectionHighCount()
JDBCConnectionPoolRuntimeMBean.getWaitSecondsHighCount()
```

The `JDBCConnectionPoolRuntimeMBean` provides methods for getting the current state of the connection pool and for getting statistics about the connection pool, such as the average number of active connections, the current number of active connections, the highest number of active connections, and so forth.

The `getState()` method returns the current state of the connection pool. The current state can be:

- `Running` if the pool is enabled (deployed and not suspended). This is the normal state of the connection pool.
- `Suspended` if the pool is disabled.
- `Shutdown` if the pool is shutdown and all database connections have been closed.
- `Unknown` if the pool state is unknown.
- `Unhealthy` if all connections are unavailable (not because they are in use). This state occurs if the database server is unavailable when the connection pool is created (creation retry must be enabled) or if all connections have failed connection tests (on creation, on reserve, on release, or periodic testing).

For more information about methods for getting connection pool statistics, see the Javadoc for the JDBCConnectionPoolRuntimeMBean at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/management/runtime/JDBCConnectionPoolRuntimeMBean.html`. Also see "Testing Connection Pools and Database Connections" on page 2-21.

Enabling Connection Creation Retries

```
JDBCConnectionPoolMBean.setConnectionCreationRetryFrequencySeconds(int seconds)
```

The `setConnectionCreationRetryFrequencySeconds()` method sets the number of seconds between attempts to create database connections when the connection pool is created. If you do not set this value, connection pool creation fails if the database is unavailable. If set and if the database is unavailable when the connection pool is created, WebLogic Server will attempt to create connections in the pool again after the number of seconds you specify, and will continue to attempt to create the connections until it succeeds.

By default, this attribute is set to `0`, which disables connection creation retries.

Note: Do not use connection creation retries with connection pools in a High Availability MultiPool. Connection requests to the MultiPool will fail (not fail-over) when a connection pool in the list is dead and the number of connection requests equals the number of connections in the first connection pool, even if connections are available in subsequent connection pools in the MultiPool.

Initializing Connections with a SQL Query

```
JDBCConnectionPoolMBean.setInitSQL(java.lang.String string)
```

With the `setInitSQL()` method, you set a value for the `initSQL` MBean attribute. WebLogic Server runs this SQL code whenever it creates a database connection for the connection pool, which includes at server startup, when expanding the connection pool, when deploying the connection pool on a server, and when refreshing a connection. In essence, WebLogic Server "primes" the connection with this SQL code before applications can use the connection. You can use this feature to set DBMS-specific operational settings that are connection-specific or to ensure that a connection has memory or permissions to perform required actions.

Start the code with `SQL` followed by a space. For example:

```
SQL alter session set NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS'
```

or

```
SQL SET LOCK MODE TO WAIT
```

Options that you can set using `InitSQL` vary by DBMS.

For information about setting this attribute with the Administration Console, see "Initializing Database Connections with SQL Code" and "Init SQL" in the *Administration Console Online Help*.

Note: `InitSQL` is not a dynamic attribute. When you change the value for `InitSQL`, you must either undeploy and redeploy the connection pool or restart the server.

Testing Connection Pools and Database Connections

```
JDBCConnectionPoolRuntimeMBean.testPool()
JDBCConnectionPoolMBean.setTestConnectionsOnCreate(boolean enable)
JDBCConnectionPoolMBean.setTestConnectionsOnRelease(boolean enable)
JDBCConnectionPoolMBean.setTestConnectionsOnReserve(boolean enable)
JDBCConnectionPoolMBean.setTestFrequencySeconds(int seconds)
JDBCConnectionPoolMBean.setTestTableName(java.lang.String table)
JDBCConnectionPoolMBean.setHighestNumUnavailable(int count)
```

To make sure that the database connections in a connection pool remain healthy, you should periodically test the connections. WebLogic Server includes two basic types of testing: *automatic* testing that you configure with attributes on the `JDBCConnectionPoolMBean` (the configuration MBean) and *manual* testing that you can do to trouble-shoot a connection pool with the `testPool()` method on the `JDBCConnectionPoolRuntimeMBean` (the runtime MBean).

Allowing WebLogic Server to automatically maintain the integrity of pool connections should prevent most DBMS connection problems. You use the following methods on the `JDBCConnectionPoolMBean` to configure automatic connection testing:

- `setTestFrequencySeconds(int seconds)`—Use this method to enable periodic connection testing and to specify the number of seconds between tests of unused connections. The server tests unused connections and reopens any faulty connections. If you do not set `TestFrequencySeconds`, periodic connection testing is not enabled. You must also set the `HighestNumUnavailable` and `TestTableName`.
- `setTestConnectionsOnCreate(boolean enable)`—Use this method to enable testing on each database connection after it is created. This applies to connections created at server startup, when the connection pool is expanded, and when a connection is recreated after failing a test. You must also set a `TestTableName`.
- `setTestConnectionsOnReserve(boolean enable)`—Use this method to enable testing on each connection before it is given to a client. This may add a slight delay to the connection request, but it guarantees that the connection is healthy. You must also set a `TestTableName`.
- `setTestConnectionsOnRelease(boolean enable)`—Use this method to enable testing on database connections when they are returned to the connection pool. You must also set a `TestTableName`.
- `setHighestNumUnavailable(int count)`—Use this method to limit the number of idle connections that the server will test. For example, if you have 10 connections in your

connection pool and 5 are in use, if the server were to begin testing all 5 connections that are not in use, there would be no connections available to fill a connection request from an application. If you set the `HighestNumUnavailable` attribute to `3`, the connection pool maintenance thread would take 3 connections from the connection pool for testing, and there would still be 2 connections available to fill a connection request.

- `setTestTableName(java.lang.String table)`—Use this method to specify a table name to use for connection testing. You can also specify SQL code to run in place of the standard test by entering `SQL` followed by a space and the SQL code you want to run as a test. `TestTableName` is required to enable any automatic database connection testing.

For information about setting these attributes in the Administration Console, see "Connection Testing Options" and "JDBC Connection Pool --> Configuration --> Connections" in the *Administration Console Online Help*.

Enabling Connection Requests to Wait for a Connection

The `JDBCConnectionPoolMBean` has two attributes that you can set to enable connection requests to wait for a connection from a connection pool: `ConnectionReserveTimeoutSeconds` and `HighestNumWaiters`. You use these two attributes together to enable connection requests to wait for a connection without disabling your system by blocking too many threads.

Connection Reserve Timeout

`JDBCConnectionPoolMBean.setConnectionReserveTimeoutSeconds(int seconds)`

When an application requests a connection from a connection pool, if all connections in the connection pool are in use and if the connection pool has expanded to its maximum capacity, the application will get a Connection Unavailable SQL Exception. To avoid this, you can configure a Connection Reserve Timeout value (in seconds) so that connection requests will wait for a connection to become available. After the Connection Reserve Timeout has expired, if no connection becomes available, the request will fail and the application will get a `PoolLimitSQLException` exception.

- If you set Connection Reserve Timeout to `0`, a connection request will wait indefinitely. Setting Connection Reserve Timeout to -1 will cause the connection to timeout immediately.
- The default value is 10 seconds.

- If you use global transactions, the Connection Reserve Timeout value is ignored. In this situation, a connection request automatically waits for a connection if one is not available. The amount of time available for an application to wait for a connection (if necessary), connect, and complete the transaction is determined by the transaction time out value.

Limiting the Number of Waiting Connection Requests

```
JDBCConnectionPoolMBean.setHighestNumWaiters(int count)
```

Connection requests that wait for a connection block a thread. If too many connection requests concurrently wait for a connection and block threads, your system performance can degrade. To avoid this, you can set the `HighestNumWaiters` attribute, which limits the number connection requests that can concurrently wait for a connection.

If you set `HighestNumWaiters` to `MAX-INT` (the default), there is effectively no bound on how many connection requests can wait for a connection. If you set `HighestNumWaiters` to `0`, connection requests cannot wait for a connection.

Configuring and Managing the Statement Cache for a Connection Pool

For each connection in a connection pool in your system, WebLogic Server creates a statement cache. When a prepared statement or callable statement is used on a connection, WebLogic Server caches the statement so that it can be reused. Statement caching is controlled by the `StatementCacheSize` and the `StatementCacheType`. For more information about how the statement cache works and configuration options, see "Increasing Performance with the Statement Cache" in the *WebLogic Server Administration Console Online Help* at `http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_connection_pools.html#statementcache`.

Each connection in the connection pool has its own statement cache, but configuration settings are made for all connections in the connection pool.

Configuring the Statement Cache

```
JDBCConnectionPoolMBean.setStatementCacheSize(int cacheSize)
JDBCConnectionPoolMBean.setStatementCacheType(java.lang.String type)
```

WebLogic Server provides methods to set the size (`StatementCacheSize`) and algorithm (`StatementCacheType`) of the statement cache for each connection pool.

When you set the StatementCacheSize, that number of statements (prepared and callable) are cached for each connection in the connection pool.

By default, the StatementCacheType is set to LRU for Least Recently Used. With this algorithm, the connection pool replaces the least recently used statement in the cache when a new prepared or callable statement is used. In most cases, this option provides the best performance. You can also set the StatementCacheType to Fixed. With the fixed algorithm, prepared and callable statements are cached until the StatementCacheSize value is met. Statements remain in the cache until the cache is cleared manually or the connection is closed.

Note: StatementCacheType is not a dynamic attribute. When you change the value for StatementCacheType, you must either undeploy and redeploy the connection pool or restart the server.

Deprecated Statement Cache Configuration Options

In releases before WebLogic Server 8.1, there were separate statement cache implementations for XA and non-XA JDBC connection pools (connection pools that use an XA JDBC driver and connection pools that use a non-XA JDBC driver to create database connections). In WebLogic Server 8.1, the statement cache was rewritten. There is now one statement cache implementation for both XA and non-XA connection pools. With the statement cache revision, there are connection pool attributes in the JDBCConnectionPoolMBean for configuring the statement cache that are now deprecated. Table 2-1 lists the deprecated MBean attributes from previous releases and the equivalent option in WebLogic Server 8.1.

Table 2-1 Deprecated Statement Cache Attributes and Equivalents

Deprecated MBean Attribute	Equivalent in WebLogic Server 8.1
PreparedStatementCacheSize	StatementCacheSize
XAPreparedStatementCacheSize	StatementCacheSize

To enable migration of a WebLogic Server configuration from an earlier release to version 8.1, Weblogic Server enforces the following order of precedence for these MBean attributes:

1. PreparedStatementCacheSize
2. XAPreparedStatementCacheSize
3. StatementCacheSize

For example, if the PreparedStatementCacheSize for a JDBC connection pool is set to 5 and the StatementCacheSize is set to 10, the actual statement cache size for each connection in the connection pool will be 5 because PreparedStatementCacheSize takes precedence over StatementCacheSize.

Clearing the Statement Cache for a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.clearStatementCache()
```

You can manually clear the statement cache for all connections in a connection pool with the clearStatementCache() method.

Clearing the Statement Cache for a Single Connection

```
weblogic.jdbc.extensions.WLConnection.clearStatementCache()
weblogic.jdbc.extensions.WLConnection.clearCallableStatement(java.lang.
String sql)
weblogic.jdbc.extensions.WLConnection.clearCallableStatement(java.lang.
String sql,int resSetType,int resSetConcurrency)
weblogic.jdbc.extensions.WLConnection.clearPreparedStatement(java.lang.
String sql)
weblogic.jdbc.extensions.WLConnection.clearPreparedStatement(java.lang.
String sql,int resSetType,int resSetConcurrency)
```

You can use methods in the weblogic.jdbc.extensions.WLConnection interface to clear the statement cache for a single connection or to clear an individual statement from the cache. These methods return true if the operation was successful and false if the operation fails because the statement was not found.

When prepared and callable statements are stored in the cache, they are stored (keyed) based on the exact SQL statement and result set parameters (type and concurrency options), if any. When clearing an individual prepared or callable statement, you must use the method that takes the proper result set parameters. For example, if you have callable statement in the cache with resSetType of ResultSet.TYPE_SCROLL_INSENSITIVE and a resSetConcurrency of ResultSet.CONCUR_READ_ONLY, you must use the method that takes the result set parameters:

```
clearCallableStatement(java.lang.String sql,int resSetType,int
resSetConcurrency)
```

If you use the method that only takes the SQL string as a parameter, the method will not find the statement, nothing will be cleared from the cache, and the method will return false.

When you clear a statement that is currently in use by an application, WebLogic Server removes the statement from the cache, but does not close it. When you clear a statement that is not currently in use, WebLogic Server removes the statement from the cache and closes it.

For more details about these methods, see the Javadoc for WLConnection at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/jdbc/extensions/WLConnection.html`.

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 enable automatic shrinking on the connection pool or manually shrink the connection pool with the `shrink()` method.

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.

Resetting a Connection Pool

`JDBCConnectionPoolRuntimeMBean.reset()`

The `JDBCConnectionPoolRuntimeMBean.reset()` method closes and reopens all 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.

Suspending a Connection Pool

`JDBCConnectionPoolRuntimeMBean.suspend()`

`JDBCConnectionPoolRuntimeMBean.forceSuspend()`

WebLogic server includes two methods in the `JDBCConnectionPoolRuntimeMbean` to suspend a connection pool: `suspend()` and `forceSuspend()`. You can use these methods to temporarily disable a connection pool, preventing any clients from obtaining or using a connection from the pool. Only users with the proper permissions can suspend a connection pool.

When you suspend a connection pool with the `suspend()` method, the connection pool is marked as disabled and applications cannot use connections from the pool. Applications that already have a reserved connection from the connection pool when it is suspended will get an exception when trying to use the connection. WebLogic Server preserves all connections in the connection pool exactly as they were before the connection pool was suspended.

When you suspend a connection pool with the `forceSuspend()` method, WebLogic Server marks the connection pool as disabled, forcibly disconnects applications that are currently using a connection, and recreates (closes and reopens) connections that were in use when the connection pool was suspended. Any transaction on the connections that are closed are rolled back. WebLogic Server preserves all other connections exactly as they were before the connection pool was suspended.

The `suspend()` and `forceSuspend()` methods replace the `disableFreezingUsers()` and `disableDroppingUsers()` methods, which are deprecated.

Resuming a Connection Pool

`JDBCConnectionPoolRuntimeMBean.resume()`

To re-enable a connection pool that you disabled with the `suspend()` or `forceSuspend()` method, you can use the `resume()` method, which marks the connection pool as enabled and allows applications to use connections from the connection pool. If you suspended the connection pool with the `suspend()` method, all connections are preserved exactly as they were before the connection pool was suspended. Clients that had reserved a connection before the connection pool was suspended can continue JDBC operations exactly where they left off. If you suspended the connection pool with the `forceSuspend()` method, connections that were not in use when the connection pool was suspended are preserved exactly as they were before the suspension. Connections that *were* in use were closed and reopened. Clients that had reserved a connection no longer have a valid JDBC context.

The `resume()` method replaces the `enable()` method, which is deprecated.

Note: You cannot use the `resume()` method to start a connection pool that did not start correctly, for example, if the database server is unavailable.

Configuring and Using 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:

- XML Application Scoping in *Programming WebLogic XML* at `http://e-docs.bea.com/wls/docs81/xml/xml_appscop.html`.
- Two-Phase Deployment Protocol in *Deploying WebLogic Server Applications* at `http://e-docs.bea.com/wls/docs81/deployment/concepts.html#two_phase`.

Configuring Application-Scoped Connection Pools

To configure an application-scoped connection pool, you add a jdbc-connection-pool element with connection pool configuration parameters to the `weblogic-application.xml` file for your enterprise application. For example:

```
<jdbc-connection-pool>
      <data-source-name>XA_LocalDS1</data-source-name>
      <connection-factory>
        <factory-name>XA_LocalCF1</factory-name>
        <connection-properties>
          <user-name>SCOTT</user-name>
          <password>tiger</password>
          <url>jdbc:oracle:thin:@dbserver:1521:sid</url>
          <driver-class-name>oracle.jdbc.xa.client.OracleXADataSource
            </driver-class-name>
          <connection-params>
            <parameter>
              <param-name>foo</param-name>
              <param-value>xyz</param-value>
            </parameter>
            <parameter>
              <param-name>bar</param-name>
```

```
        <param-value>abc</param-value>
      </parameter>
    </connection-params>
  </connection-properties>
</connection-factory>

<pool-params>
  <size-params>
    <initial-capacity>5</initial-capacity>
    <max-capacity>10</max-capacity>
    <capacity-increment>2</capacity-increment>
    <shrinking-enabled>true</shrinking-enabled>
    <shrink-frequency-seconds>300</shrink-frequency-seconds>
    <highest-num-waiters>100</highest-num-waiters>
    <highest-num-unavailable>4</highest-num-unavailable>
  </size-params>

  <xa-params>
    <debug-level>3</debug-level>
    <local-transaction-supported>true</local-transaction-supported>
    <xa-set-transaction-timeout>true</xa-set-transaction-timeout>
    <xa-transaction-timeout>30</xa-transaction-timeout>
  </xa-params>

  <login-delay-seconds>1</login-delay-seconds>
  <leak-profiling-enabled>false</leak-profiling-enabled>
  <connection-check-params>
    <table-name>check_table</table-name>
    <check-on-create-enabled>true</check-on-create-enabled>
    <check-on-reserve-enabled>true</check-on-reserve-enabled>
    <check-on-release-enabled>false</check-on-release-enabled>
    <connection-reserve-timeout-seconds>30
      </connection-reserve-timeout-seconds>
    <test-frequency-seconds>600</test-frequency-seconds>
    <connection-creation-retry-frequency-seconds>360
      </connection-creation-retry-frequency-seconds>
    <inactive-connection-timeout-seconds>360
      </inactive-connection-timeout-seconds>
    <init-sql>SQL SET LOCK MODE TO WAIT</init-sql>
  </connection-check-params>
```

```
    </pool-params>
    <driver-params>
      <prepared-statement>
        <cache-size>10</cache-size>
        <cache-type>LRU</cache-type>
      </prepared-statement>
      <row-prefetch-enabled>true</row-prefetch-enabled>
      <row-prefetch-size>500</row-prefetch-size>
      <stream-chunk-size>1024</stream-chunk-size>
    </driver-params>
  </jdbc-connection-pool>
```

For more details about JDBC connection pool element entries, see weblogic-application.xml Deployment Descriptor Elements in *Developing WebLogic Server Applications* at `http://e-docs.bea.com/wls/docs81/programming/app_xml.html#app-scoped-pool`.

If you deploy your enterprise application as an exploded archive, you can also change configuration options using the Administration Console. See "Application-Scoped JDBC Data Sources and Connection Pools" in the *Administration Console Online Help* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_connection_pools.html#app_scoped_pools.

Required Elements Within the jdbc-connection-pool Element

When configuring and application-scoped connection pool within the `weblogic-application.xml` file, you must include the following sub-elements:

- `data-source-name`, which defines a name for the application-scoped data source created (always a TxDataSource) with the application-scoped connection pool when you deploy your application. The application uses this name to look up the data source on the local JNDI tree to get a connection from the connection pool.

  ```
  <data-source-name>XA_LocalDS1</data-source-name>
  ```

  See "Getting a Connection from an Application-Scoped Connection Pool" on page 2-35 for more information.

- `connection-factory`, which is a reference to the data source factory in your WebLogic domain to use to create the application-scoped data source and connection pool when you deploy your application. The data source factory also supplies some default values for

connections in the application-scoped connection pool. You can over-ride these values. For example:

```
<connection-factory>
  <factory-name>XA_LocalCF1</factory-name>
  <connection-properties>
    <user-name>SCOTT</user-name>
    <password>tiger</password>
    <url>jdbc:oracle:thin:@dbserver:1521:sid</url>
    <driver-class-name>oracle.jdbc.xa.client.OracleXADataSource
      </driver-class-name>
    <connection-params>
      <parameter>
        <param-name>foo</param-name>
        <param-value>xyz</param-value>
      </parameter>
      <parameter>
        <param-name>bar</param-name>
        <param-value>abc</param-value>
      </parameter>
    </connection-params>
  </connection-properties>
</connection-factory>
```

If you do not specify a data source factory name, you must provide all parameters necessary to create the connection pool, including the user name, password, URL, driver class name, and connection parameters in the `connection-properties` tag.

For more information about configuring a data source factory in your WebLogic domain, see "JDBC Data Source Factories" in the *Administration Console Online Help* at `http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_datasources.html#dsfact`.

Encrypting the Database Password in weblogic-application.xml

To avoid storing or transmitting database passwords in clear text, you can encrypt database passwords in the `weblogic-application.xml` file with the `weblogic.j2ee.PasswordEncrypt` utility. This utility searches for database passwords in the following places:

- In the password tag:

```
<connection properties>
  <password>tiger</password>
</connection properties>
```

- In the connection-params tag:

```
<connection properties>
  <parameter>
     <param-name>password</param-name>
     <param-value>tiger</param-value>
  </parameter>
</connection properties>
```

The utility hashes the passwords, replaces the passwords in the `weblogic-application.xml` file with a hashed version, and stores the hashed values in the `SerializedSystemIni.dat` in your WebLogic domain.

Note: Password encryption is *domain specific*. That is, when you run the encryption utility, you must specify the domain in which you will deploy your application. If you try to deploy the application in another domain, WebLogic Server will not be able to decrypt the passwords for use at runtime. For more information about encrypting passwords, see "Protecting Passwords" in *Managing WebLogic Security* at `http://e-docs.bea.com/wls/docs81/secmanage/passwords.html#protect_passwords`.

You run this utility before your application archive is created. You cannot run it on a file that is already archived.

Before you run this utility, you should have WebLogic Server installed and your environment configured (so that the utility can find required classes). The server does not have to running when you run the password encryption utility.

To run the password encryption utility, enter the following command:

```
java weblogic.j2ee.PasswordEncrypt <descriptor file> <domain config dir>
```

Where:

- `descriptor file` is the weblogic-application.xml for the application.
- `domain config dir` is the root directory of the WebLogic domain (which contains the config.xml file).

After you run the password encryption utility, passwords may look like:

- In the password tag:

```
<connection properties>
  <password>{3DES}iaHh5dH7clU=</password>
</connection properties>
```

- In the connection-params tag:

```
<connection properties>
  <parameter>
     <param-name>password</param-name>
     <param-value>{3DES}iaHh5dH7clU=</param-value>
  </parameter>
</connection properties>
```

Notes: If you need to change a password, you can change it in the weblogic-application.xml file and then re-run the password encryption utility. The utility will not re-encrypt passwords that are already encrypted.

You must re-encrypt passwords in the descriptor file if:

- You move the application from one installation of WebLogic Server to another.
- You delete the domain directory referenced when encrypting passwords, even if the directory is recreated.

Deprecated Statement Cache Configuration Options for Application-Scoped Connection Pools

In releases before WebLogic Server 8.1, there were separate statement cache implementations for XA and non-XA JDBC connection pools. In WebLogic Server 8.1, the statement cache was rewritten. There is now one statement cache implementation for both XA and non-XA connection pools. With the statement cache revision, there is one tag available in the weblogic-application.xml descriptor file that is deprecated. Table 2-2 lists the deprecated descriptor tag, its replacement, and the related MBean attributes created when the application-scoped connection pool is deployed.

Table 2-2 Deprecated Statement Cache Descriptor Tags and Related MBeans Attributes

Deprecated	Equivalent in WebLogic Server 8.1

Deprecated descriptor tag:
`<pool-params>`
`<xa-params>`
`<prepared-statement-cache-size>10`
`</prepared-statement-cache-size>`
`</xa-params>`
`</pool-params>`
Note: Only the tag in bold is deprecated. The other tags are listed for contextual purposes only.
	Use this tag instead:
	`<driver-params>`
	`<prepared-statement>`
	`<cache-size>10`
	`</cache-size>`
	`</prepared-statement>`
	`</driver-params>`

MBean attribute set from tag above: `XaParamsMBean.PreparedStatementCacheSize`	MBean attribute set from tag above: `PreparedStatementMBean.CacheSize`

To enable migration of a WebLogic Server configuration or enterprise application from an earlier release to version 8.1, Weblogic Server enforces the following order of precedence for these MBean attributes:

1. `PreparedStatementMBean.CacheSize`
2. `XAParamsMBean.PreparedStatementCacheSize`

For example, if the `<cache-size>` for a JDBC connection pool is set to `5` in the `weblogic-application.xml` file and the `<prepared-statement-cache-size>` is set to `10`, the actual statement cache size for each connection in the connection pool will be 5 because `PreparedStatementMBean.CacheSize` takes precedence over `XaParamsMBean.PreparedStatementCacheSize`.

Note: When migrating an application from WebLogic Server 7.0 SP3 or later, to disable XA statement caching, you must set the `<cache-size>` for the JDBC connection pool in the `weblogic-application.xml` file to `0`.

Getting a Connection from an Application-Scoped Connection Pool

To get a connection from an application-scoped connection pool, you look up the data source defined in the weblogic-application.xml descriptor file in the local environment (`java:comp/env`) and then request a connection from the data source. For example:

```
javax.sql.DataSource ds =
    (javax.sql.DataSource) ctx.lookup("java:app/jdbc/myDataSource");
java.sql.Connection conn = ds.getConnection();
```

When you are finished using the connection, make sure you close the connection to return it to the connection pool:

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

Configuring and Using MultiPools

A MultiPool is a pool of connection pools. 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.

Configuring MultiPools

MultiPools contain a configurable algorithm for choosing which connection pool will return a connection to the client.

Create a MultiPool using the following steps:

1. Create necessary connection pools.
2. Determine if the primary purpose of the MultiPool is high availability or load balancing. See "Choosing the MultiPool Algorithm" on page 2-36.
3. Create the MultiPool using the Administration Console or WebLogic Management API and assign the connection pools to the MultiPool.

For more information about MultiPools, see the Administration Console Online Help at `http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_multipools.html`. For information about the `JDBCMultiPoolMBean`, see the WebLogic Server Javadocs at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/management/configuration/JDBCMultiPoolMBean.html`.

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.

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 test fails and the connection cannot be replaced, or if the connection pool is suspended, a connection is sought sequentially from the next pool on the list.

Note: This algorithm relies on `TestConnectionsOnReserve` to test to see if a connection in the first connection pool is healthy. If the connection fails the test, the MultiPool uses a connection from the next connection pool in the MultiPool. See "Testing Connection Pools and Database Connections" on page 2-21 for information about configuring `TestConnectionsOnReserve`.

Load Balancing

Connection requests to a load balancing MultiPool are served from any connection pool in the list. Pools are accessed using a round-robin scheme. When the MultiPool provides a connection, it selects a connection from the connection pool listed just after the last pool that was used to provide a connection. MultiPools that use the Load Balancing algorithm also fail over to the next connection pool in the list if a database connection test fails and the connection cannot be replaced, or if the connection pool is suspended.

Transaction Support in JDBC MultiPools

In WebLogic Server 8.1SP5, MultiPools were enhanced to provide support for global transactions.

Note: WebLogic Server 8.1 SP5 is certified to support Multipools with XA only on Oracle RAC. For information on supported versions of Oracle RAC, see Supported Database Configurations.

For an example of a MultiPool configuration that supports global transactions, see "Using MultiPools with Global Transactions".

Transaction Failover Processing for MultiPools

If a connection from a MultiPool fails while a global transaction is in progress, the result of the transaction depends on the stage of the transaction at the time of the connection failure.

The first stage at which a failure may occur is before the application calls for the transaction to be committed. If a database connection fails at this stage, the application gets an exception and must get a new connection and make a new attempt at processing the transaction. WebLogic Server does not support transparent failover.

If a failure occurs after the application has called for the transaction to be committed, the handling of any in-flight transaction depends upon whether the `PREPARE` operation is complete. If the `PREPARE` operation is not complete, the transaction manager rolls back the transaction and sends the application an exception for the failed transaction. If the `PREPARE` operation is complete, the transaction manager attempts to drive the in-flight transaction to completion using another connection.

If a failure occurs during the `COMMIT` operation, the transaction manager attempts to retry the `COMMIT` operation several times, depending on the `XARetryDurationSeconds` setting. Note that the connection is blocked during these attempts. If the `COMMIT` operation is not successful during the first set of retry attempts, the application gets an exception. The transaction manager then continues to retry the `COMMIT` operation periodically until it is successful or until it reaches the JTA Abandon Timeout period defined in the configuration file and abandons the transaction.

MultiPool Failover Enhancements

In WebLogic Server 8.1SP3, the following enhancements were made to MultiPools:

- Connection request routing enhancements to avoid requesting a connection from an automatically disabled (dead) connection pool within a MultiPool. See "Connection Request Routing Enhancements When a Connection Pool Fails."
- Automatic failback on recovery of a failed connection pool within a MultiPool. See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool."
- Failover for busy connection pools within a MultiPools. See "Enabling Failover for Busy Connection Pools in a MultiPool."
- Failover callbacks for MultiPools with the High Availability algorithm. See "Controlling MultiPool Failover with a Callback."
- Failback callbacks for MultiPools with either algorithm. See "Controlling MultiPool Failback with a Callback."

Connection Request Routing Enhancements When a Connection Pool Fails

To improve performance when a connection pool within a MultiPool fails, WebLogic Server automatically disables the connection pool when a pooled connection fails a connection test. After a connection pool is disabled, WebLogic Server does not route connection requests from applications to the connection pool. Instead, it routes connection requests to the next available connection pool listed in the MultiPool.

This feature requires that connection pool testing options are configured for all connection pools in a MultiPool, specifically `TestTableName` and `TestConnectionsOnReserve`.

If a callback handler is registered for the MultiPool, WebLogic Server calls the callback handler before failing over to the next connection pool in the list. See "Controlling MultiPool Failover with a Callback" on page 2-40 for more details.

Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool

After a connection pool is automatically disabled because a connection failed a connection test, WebLogic Server periodically tests a connection from the disabled connection pool to determine when the connection pool (or underlying database) is available again. When the connection pool becomes available, WebLogic Server automatically re-enables the connection pool and resumes routing connection requests to the connection pool, depending on the MultiPool algorithm and the position of the connection pool in the list of included connection pools.

To control how often WebLogic Server checks automatically disabled connection pools in a MultiPool, you add a value for the `HealthCheckFrequencySeconds` attribute to the MultiPool configuration in the `config.xml` file. For example:

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="240"
Targets="examplesServer" />
```

Note: This attribute is not available in the administration console. To implement this functionality, you must manually add the attribute to the MultiPool configuration in the `config.xml` file.

WebLogic Server waits for the period you specify between connection tests for each disabled connection pool. The default value is 300 seconds. If you do not specify a value, WebLogic Server will test automatically disabled connection pools every 300 seconds.

This feature requires that connection pool testing options are configured for all connection pools in a MultiPool, specifically `TestTableName` and `TestConnectionsOnReserve`.

WebLogic Server does not test and automatically re-enable connection pools that you manually disable. It only tests connection pools that it automatically disables.

If a callback handler is registered for the MultiPool, WebLogic Server calls the callback handler before re-enabling the connection pool. See "Controlling MultiPool Failback with a Callback" on page 2-42 for more details.

Enabling Failover for Busy Connection Pools in a MultiPool

By default, for MultiPools with the High Availability algorithm, when the number of requests for a database connection exceeds the number of available connections in the current connection pool in the MultiPool, subsequent connection requests fail.

To enable the MultiPool to failover when all connections in the current connection pool are in use, you must set a value for the `FailoverRequestIfBusy` attribute in the MultiPool configuration in the `config.xml` file. If set to `true`, when all connections in the current connection pool are in use, application requests for connections will be routed to the next available connection pool within the MultiPool. When set to `false` (the default), connection requests do not failover. Weblogic Server throws a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

After you add the `FailoverRequestIfBusy` attribute to the `config.xml` file, the MultiPool entry may look like the following:

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
PoolList="demoPool2,demoPool"
FailoverRequestIfBusy="true"
Targets="examplesServer" />
```

Note: The `FailoverRequestIfBusy` attributes is not available in the administration console. To implement this functionality, you must manually add this attribute to the MultiPool configuration in the `config.xml` file.

If a `ConnectionPoolFailoverCallbackHandler` is included in the MultiPool configuration, WebLogic Server calls the callback handler before failing over. See "Controlling MultiPool Failover with a Callback" on page 2-40 for more details.

Controlling MultiPool Failover with a Callback

You can register a callback handler with WebLogic Server that controls when a MultiPool with the High-Availability algorithm fails over connection requests from one JDBC connection pool in the MultiPool to the next connection pool in the list.

You can use callback handlers to control if or when the failover occurs so that you can make any other system preparations before the failover, such as priming a database or communicating with a high-availability framework.

Callback handlers are registered via an attribute of the MultiPool in the `config.xml` file and are registered per MultiPool. Therefore, you must register a callback handler for each MultiPool to which you want the callback to apply. And you can register different callback handlers for each MultiPool.

Callback Handler Requirements

A callback handler used to control the failover and failback within a MultiPool must include an implementation of the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface. When the MultiPool needs to failover to the next connection pool in the list or when a previously disabled connection pool becomes available, WebLogic Server calls the `allowPoolFailover()` method in the `ConnectionPoolFailoverCallback` interface, and passes a value for the three parameters, `currPool`, `nextPool`, and `opcode`, as defined below. WebLogic Server then waits for the return from the callback handler before completing the task.

Your application must return OK, RETRY_CURRENT, or DONOT_FAILOVER as defined below.The application should handle failover and failback cases.

See the Javadoc for the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface for more details.

Note: Failover callback handlers are optional.If no callback handler is specified in the MultiPool configuration, WebLogic Server proceeds with the operation (failing over or re-enabling the disabled connection pool).

Callback Handler Configuration

There are two MultiPool configuration attributes associated with the failover and failback functionality:

- `ConnectionPoolFailoverCallbackHandler`—To register a failover callback handler for a MultiPool, you add a value for this attribute to the MultiPool configuration in the `config.xml` file. The value must be an absolute name, such as `com.bea.samples.wls.jdbc.MultiPoolFailoverCallbackApplication`.

`HealthCheckFrequencySeconds`—To control how often WebLogic Server checks disabled (dead) connection pools in a MultiPool to see if they are now available, you can add a value for this attribute to the MultiPool configuration in the `config.xml` file.

The maximum value that can be passed to the method is MAXINT while the minimum value is 0. Setting the value to zero disables the attribute.See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool" on page 2-38 for more details.

After you add the attributes to the `config.xml` file, the MultiPool entry may look like the following:

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
ConnectionPoolFailoverCallbackHandler="com.bea.samples.wls.jdbc.MultiPoolFailoverCallbackApplication"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="120"
Targets="examplesServer" />
```

Note: These attributes are not available in the administration console. To implement this functionality, you must manually add these attributes to the MultiPool configuration in the `config.xml` file.

How It Works—Failover

WebLogic Server attempts to failover connection requests to the next connection pool in the list when the current connection pool fails a connection test or, if you enabled `FailoverRequestIfBusy`, when all connections in the current connection pool are busy.

To enable the callback feature, you register the callback handler with Weblogic Server using the `ConnectionPoolFailoverCallbackHandler` attribute in the MultiPool configuration in the `config.xml` file.

With the High Availability algorithm, connection requests are served from the first connection pool in the list. If a connection from that connection pool fails a connection test, WebLogic Server marks the connection pool as dead and disables it. If a callback handler is registered, WebLogic Server calls the callback handler, passing the following information, and waits for a return:

- `currPool`—For failover, this is the name of connection pool currently being used to supply database connections. This is the "failover from" connection pool.

- `nextPool`—The name of next available connection pool listed in the MultiPool. For failover, this is the "failover to" connection pool.
- `opcode`—A code that indicates the reason for the call:
  - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current connection pool is dead and has disabled it.
  - `OPCODE_CURR_POOL_BUSY`—All database connections in the connection pool are in use. (Requires `FailoverIfBusy=true` in the MultiPool configuration. See "Enabling Failover for Busy Connection Pools in a MultiPool" on page 2-39.)

Failover is synchronous with the connection request: Failover occurs only when WebLogic Server is attempting to satisfy a connection request.

The return from the callback handler can indicate one of three options:

- `OK`—proceed with the operation. In this case, that means to failover to the next connection pool in the list.
- `RETRY_CURRENT`—Retry the connection request with the current connection pool.
- `DONOT_FAILOVER`—Do not retry the current connection request and do not failover. WebLogic Server will throw a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

WebLogic Server acts according to the value returned by the callback handler.

If the secondary connection pools fails, WebLogic Server calls the callback handler again, as in the previous failover, in an attempt to failover to the next available connection pool in the MultiPool, if there is one.

Note: WebLogic Server does *not* call the callback handler when you manually disable a connection pool.

For MultiPools with the Load-Balancing algorithm, WebLogic Server does not call the callback handler when a connection pool is disabled. However, it does call the callback handler when attempting to re-enable a disabled connection pool. See the following section for more details.

Controlling MultiPool Failback with a Callback

If you register a failover callback handler for a MultiPool, WebLogic Server calls the same callback handler when re-enabling a connection pool that was automatically disabled. You can use the callback to control if or when the disabled connection pool is re-enabled so that you can make any other system preparations before the connection pool is re-enabled, such as priming a database or communicating with a high-availability framework.

Callback handlers are registered via an attribute of the MultiPool in the `config.xml` file and are registered per MultiPool. Therefore, you must register a callback handler for each MultiPool to which you want the callback to apply. And you can register different callback handlers for each MultiPool.

See the following sections for more details about the callback handler:

- "Callback Handler Requirements" on page 2-40
- "Callback Handler Configuration" on page 2-40

How It Works—Failback

WebLogic Server periodically checks the status of connection pools in a MultiPool that were automatically disabled. (See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool" on page 2-38.) If a disabled connection pool becomes available and if a failover callback handler is registered, WebLogic Server calls the callback handler with the following information and waits for a return:

- `currPool`—For failback, this is the name of the connection pool that was previously disabled and is now available to be re-enabled.
- `nextPool`—For failback, this is null.
- `opcode`—A code that indicates the reason for the call. For failback, the code is always `OPCODE_REENABLE_CURR_POOL`, which indicates that the connection pool named in `currPool` is now available.

Failback, or automatically re-enabling a disabled connection pool, differs from failover in that failover is synchronous with the connection request, but failback is asynchronous with the connection request.

The callback handler can return one of the following values:

- `OK`—proceed with the operation. In this case, that means to re-enable the indicated connection pool. WebLogic Server resumes routing connection requests to the connection pool, depending on the MultiPool algorithm and the position of the connection pool in the list of included connection pools.
- `DONOT_FAILOVER`—Do not re-enable the `currPool` connection pool. Continue to serve connection requests from the connection pool(s) in use.

WebLogic Server acts according to the value returned by the callback handler.

If the callback handler returns DONOT_FAILOVER, WebLogic Server will attempt to re-enable the connection pool during the next testing cycle as determined by the `HealthCheckFrequencySeconds` attribute in the MultiPool configuration, and will call the callback handler as part of that process.

The order in which connection pools are listed in a MultiPool is very important. A MultiPool with the High Availability algorithm will always attempt to serve connection requests from the first available connection pool in the list of connection pools in the MultiPool. Consider the following scenario:

`MultiPool_1` uses the High Availability algorithm, has a registered `ConnectionPoolFailoverCallbackHandler`, and includes three connection pools: `CP1`, `CP2`, and `CP3`, listed in that order.

`CP1` becomes disabled, so `MultiPool_1` fails over connection requests to `CP2`.

`CP2` then becomes disabled, so `MultiPool_1` fails over connection requests to `CP3`.

After some time, `CP1` becomes available again and the callback handler allows WebLogic Server to re-enable the connection pool. Future connection requests will be served by `CP1` because `CP1` is the first connection pool listed in the MultiPool.

If `CP2` subsequently becomes available and the callback handler allows WebLogic Server to re-enable the connection pool, connection requests will continue to be served by `CP1` because `CP1` is listed before `CP2` in the list of connection pools.

MultiPool Fail-Over Limitations and Requirements

WebLogic Server provides the High Availability algorithm for MultiPools so that if a connection pool fails (for example, if the database management system crashes), your system can continue to operate. However, you must consider the following limitations and requirements when configuring your system.

Test Connections on Reserve to Enable Fail-Over

Connection pools rely on the `TestConnectionsOnReserve` feature to know when database connectivity is lost. Connections are *not* automatically tested before being reserved by an application. You must set `TestConnectionsOnReserve=true` for the connection pools within the MultiPool. After turning on this feature, WebLogic Server will test each connection before returning it to an application, which is crucial to the High Availability algorithm operation. With the High Availability algorithm, the MultiPool uses the results from testing connections on reserve to determine when to fail over to the next connection pool in the MultiPool. After a test

failure, the connection pool attempts to recreate the connection. If that attempt fails, the MultiPool fails over to the next connection pool.

By Default, No Fail-Over When All Connections are In Use

By default, if all connections in the primary connection pool are being used, a MultiPool with the High Availability algorithm will not attempt to provide a connection from the next pool in the list. MultiPool failover takes effect only if loss of database connectivity has occurred (or the connection pool has been disabled). See "Enabling Failover for Busy Connection Pools in a MultiPool" on page 2-39 for information about enabling failover in a MultiPool when all connections in a connection pool are in use.

Do Not Enable Connection Creation Retries

Do not enable connection creation retries with connection pools in a High Availability MultiPool. Connection requests to the MultiPool will fail (not fail-over) when a connection pool in the list is dead and the number of connection requests equals the number of connections in the first connection pool, even if connections are available in subsequent connection pools in the MultiPool.

MultiPools and the connection creation retries feature both attempt to solve the same problem—to gracefully handle database connections when a database is unavailable. If you use these two features together, their functionality will interfere with each other.

No Fail-Over for In-Use Connections

It is possible for a connection to fail after being reserved, in which case your application must handle the failure. WebLogic Server cannot provide fail-over for connections that fail while being used by an application. Any failure while using a connection requires that you restart the transaction and provide code to handle such a failure.

CHAPTER 3

Performance Tuning Your JDBC Application

The following sections explain how to get 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? For a typical application that will use one connection from each involved connection pool per transaction or invoke, each connection pool will need one connection for each transaction that can be running simultaneously, which is the number of execute threads the server is running. As soon as a thread is finished, it will free its connections to reuse when the thread starts the next transaction. Therefore, the pool size should

usually equal the number of execute threads. Add more if the connection pool is configured to periodically test database connections.

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/docs81/ConsoleHelp/jdbc_connection_pools.html#statementcache`)
- Read-Only Entity Beans (`http://e-docs.bea.com/wls/docs81/ejb/entity.html`)
- JNDI in a Clustered Environment (`http://e-docs.bea.com/wls/docs81/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
Mike	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 Wrapper Drivers

BEA recommends that you use DataSource objects to get database connections in new applications. DataSource objects, along with the JNDI tree, 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 wrapper drivers to get database connectivity.

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

Using the WebLogic RMI Driver

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 through DataSource objects, which are created in the Administration Console. You should create DataSource objects in your WebLogic Server configuration before you 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/docs81/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:
 }
  catch (Exception e) {
    // a failure occurred
    log message;
  }
} finally {
  try {
    ctx.close();
  } catch (Exception e) {
     log message; }
  try {
    if (rs != null) rs.close();
  } catch (Exception e) {
     log message; }
  try {
    if (stmt != null) stmt.close();
  } catch (Exception e) {
     log message; }
  try {
    if (conn != null) conn.close();
```

```
  } catch (Exception e) {
     log message; }
}
```

(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/docs81/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 global 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. Your application uses the JTS driver to access a connection from the connection pool.

WebLogic Server also uses the JTS driver internally when a connection from a connection pool that uses a non-XA JDBC driver participates in a global transaction. This behavior enables a non-XA resource to emulate XA and participate in a two-phase commit transaction. See "Configuring Non-XA JDBC Drivers for Distributed Transactions" in the *Administration Console Online Help* at `http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_connection_pools.html#confignonXA`.

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.

For the JTS driver and your application to participate in a global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` within a global transaction. After the transaction completes (gets committed or rolled back), WebLogic Server puts the connection back in the connection pool. If you want to use a connection for another global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` again within a new global 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:

```
// Start the global transaction before getting a connection
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. 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
- Third-party JDBC drivers that do not implement Serializable or Remote interfaces cannot pass objects to a remote client application.

For more information, see "Supported Database Configurations" in *Supported Configurations for WebLogic Platform 8.1*.

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

- "Using Third-Party JDBC Drivers Installed with WebLogic Server" on page 5-2
- "Using Third-Party JDBC Drivers not Installed with WebLogic Server" on page 5-2

Using Third-Party JDBC Drivers Installed with WebLogic Server

The following third-party drivers are installed with WebLogic Server:

- Oracle Thin Driver 10g and 9.2.0
- Sybase jConnect 4.5 (jConnect.jar), 5.5 (jconn2.jar), and 6.0 (jconn3.jar)

Note: JDBC Driver support changed in the following releases:

- In WebLogic Server 8.1SP3, the default version of the Oracle Thin driver was changed to the 10g driver (the version in *WL_HOME*\server\lib). In previous releases of WebLogic Server 8.1, the 9.2.0 version of the Oracle Thin driver was the default version of the driver.
- In WebLogic Server 8.1SP6, support was added for the Sybase JConnect 6.0 (JDBC 2.0) driver.

Drivers installed with Weblogic server are located 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 these files so that they are loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add these JDBC drivers to your CLASSPATH.

Using Third-Party JDBC Drivers not Installed with WebLogic Server

If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you need to update the WebLogic Server's classpath to include the location of the JDBC driver classes.

Edit the `commEnv.cmd/sh` script in *WL_HOME*`/common/bin` and prepend your classes as described in "Modifying the Classpath" in *WebLogic Server Command Reference*.

Using the Oracle Thin Driver

The following secions provide information on using the Oracle Thin Driver:

Updating the Oracle 10g Driver

If you plan to use a different version of the driver, you must replace the `ojdbc14.jar` file in *WL_HOME*`\server\lib` with an updated version of the file from Oracle or add the new file to the front of your `CLASSPATH`. You can download driver updates from the Oracle Web site at `http://otn.oracle.com/software/content.html`.

Note: The `ojdbc14.jar` file replaces `classes12.zip` as the source for Oracle Thin driver classes. This version of the driver is for use with a Java 2 SDK version 1.4.

Using the Oracle 9.2 Driver

The *WL_HOME*`\server\ext\jdbc\oracle` folder (where *WL_HOME* is the folder where WebLogic Platform is installed) of your WebLogic Server installation includes subfolders for the 9.2.0 and 10g versions of the Oracle Thin driver.

To use the 9.2.0 version of the driver:

1. In Windows Explorer or a command shell, navigate to the *WL_HOME*`\server\ext\jdbc\oracle\920` folder.
2. Copy `ojdbc14.jar`.
3. In Windows Explorer or a command shell, navigate to *WL_HOME*`\server\lib` and replace the existing version of `ojdbc14.jar` with the version you copied.

To revert to version 10g (the default), follow the instructions above, but copy from the following folder: *WL_HOME*\server\ext\jdbc\oracle\10g.

Package Change for Oracle Thin Driver 9.x and 10g

For Oracle 8.x and previous releases, the package that contained the Oracle Thin driver was `oracle.jdbc.driver`. When configuring a JDBC connection pool that uses the Oracle 8.1.7 Thin driver, you specify the DriverName (Driver Classname) as `oracle.jdbc.driver.OracleDriver`. For Oracle 9.x and 10g, the package that contains the Oracle Thin driver is `oracle.jdbc`. When configuring a JDBC connection pool that uses the Oracle 9.x or 10g Thin driver, you specify the DriverName (Driver Classname) as `oracle.jdbc.OracleDriver`. You can use the `oracle.jdbc.driver.OracleDriver` class with the 9.x and 10g drivers, but Oracle may not make future feature enhancements to that class.

See the Oracle documentation for more details about the Oracle Thin driver.

Note: The package change does not apply to the XA version of the driver. For the XA version of the Oracle Thin driver, use `oracle.jdbc.xa.client.OracleXADataSource` as the DriverName (Driver Classname) in a JDBC connection pool.

Character Set Support with nls_charset12.zip

The Oracle Thin driver includes Globalization Support for all Oracle character sets for CHAR and NCHAR datatypes not retrieved or inserted as part of an Oracle object or collection type.

However, in the case of the CHAR and VARCHAR data portion of Oracle objects and collections, the Oracle Thin driver includes Globalization Support support for only the following character sets:

- US7ASCII
- WE8DEC
- ISO-LATIN-1
- UTF-8

If you use other character sets with CHAR and NCHAR data in Oracle object types and collections, you must include `nls_charset.zip` in your `CLASSPATH`. If this file is not in your `CLASSPATH`, you will see the following exception:

```
java.sql.SQLException: Non supported character set:
oracle-character-set-178
```

The nls_charset12.zip file is installed with WebLogic Server in the *WL_HOME*\server\ext\jdbc\oracle\920 and *WL_HOME*\server\ext\jdbc\oracle\10g folders (where *WL_HOME* is the folder where WebLogic Server is installed). See "Using Third-Party JDBC Drivers not Installed with WebLogic Server" on page 5-2 for instructions to set your CLASSPATH.

Note: For Globalization Support with the 10g version of the driver, Oracle supplies the orai18n.jar file, which replaces nls_charset.zip. If you use character sets other than US7ASCII, WE8DEC, WE8ISO8859P1 and UTF8 with CHAR and NCHAR data in Oracle object types and collections, you must include orai18n.jar in your CLASSPATH. orai18n.jar is *not* installed with WebLogic Server. You can download it from the Oracle Web site.

Using the Oracle Thin Driver in Debug Mode

The *WL_HOME*\server\ext\jdbc\oracle\ (where *WL_HOME* is the folder where WebLogic Server is installed) includes subfolders for the 9.2.0 and 10g versions of the Oracle Thin driver. Each subfolder contains a ojdbc14_g.jar file, which contains the necessary classes to support debug and trace.

To use the Oracle Thin driver in debug mode:

1. Prepend the path of the ojdbc14_g.jar file to the WebLogic Server classpath as described in "Modifying the Classpath" in *WebLogic Server Command Reference*.
2. Turn on JDBC logging (see "Enabling JDBC Logging" in the *WebLogic Server Administration Console Online Help* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/logging.html#jdbc_log).

Updating the Sybase jConnect Driver

WebLogic Server ships with Sybase jConnect 4.5 (jConnect.jar), 5.5 (jconn2.jar), and 6.0 (jconn3.jar) preconfigured and ready to use. To use a different version, replace the Sybase.jar file located at *WL_HOME*\server\lib with the updated version of the file from the DBMS vendor.

To revert to versions installed with WebLogic Server, copy the following files and place them in the *WL_HOME*\server\lib folder:

- *WL_HOME*\server\ext\jdbc\sybase\jConnect.jar
- *WL_HOME*\server\ext\jdbc\sybase\jConnect-5_5\classes\jconn2.jar
- *WL_HOME*\server\ext\jdbc\sybase\jConnect-6_0\classes\jconn3.jar

Installing and Using the IBM DB2 Type 2 JDBC Driver

The IBM DB2 client installation includes a type 2 JDBC driver that you can use to create connections to a DB2 database in a connection pool. By default, the DB2 client uses a JDBC 1.x version of the driver. To use the JDBC 2.0-compliant version of the driver, follow the steps below.

Note: You must install the DB2 client on each machine that you want to use the DB2 type 2 JDBC driver to connect to the database. As with all type 2 drivers, the DB2 driver relies on libraries in the database client to access the database.

1. Stop the DB2 JDBC Applet Server Windows Service.
2. In the *db2_install_path*\java12 directory, where *db2_install_path* is the directory in which you installed the DB2 client, run the usejdbc2.bat batch file.

   This batch file creates a folder for the JDBC 1.2 version of JDBC driver and then replaces files in the *db2_install_path*\java folder with the JDBC 2.0 version of the driver.
3. Start the DB2 JDBC Applet Server Windows service.
4. Check the contents of the *db2_install_path*\java12\inuse file. If JDBC 2.0 is being used, the file will contain JDBC 2.0.

Before you can use the DB2 type 2 JDBC driver in a connection pool, you must add the driver classes to your CLASSPATH and the DB2 client libraries to your PATH. You may want to do this in the start scripts for your domain. For example:

```
set CLASSPATH=db2_install_path\java\db2java.zip;%CLASSPATH%
set PATH=db2_install_path\bin;%PATH%
```

Where *db2_install_path* is the directory in which you installed the DB2 client.

If you plan to use the XA version of the IBM DB2 driver, see "Using the IBM DB2 Type 2 JDBC Driver" in *Programming WebLogic JTA* at http://e-docs.bea.com/wls/docs81/jta/thirdpartytx.html#db2 for configuration instructions.

To create a connection pool with connections that use the DB2 type 2 driver, you can use the JDBC Connection Pool Assistant in the Administration Console (see "JDBC Connection Pools" in the *Administration Console Online Help* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_connection_pools.html) or the JMX API (see "Creating a Connection Pool Dynamically" on page 2-9).

Connection Pool Attributes when using the IBM DB2 Type 2 JDBC Driver

Use the attributes as described in Table 5-1 and Table 5-2 when creating a connection pool that uses the IBM DB2 Type 2 JDBC Driver.

Table 5-1 Non-XA Connection Pool Attributes Using the DB2 Type 2 JDBC Driver

Attribute	Value
URL	`jdbc:db2:`*`dbname`*
Driver Class Name	`COM.ibm.db2.jdbc.app.DB2Driver`
Properties	`user=`*`username`* `DatabaseName=`*`dbname`*
Password	*`password`*
TestConnectionsOnCreate	`true`
TestConnectionsOnReserve	`true`
TestTableName	`SYSIBM.SYSTABLES`
Target	`serverName`

The database name in the URL and in the Properties string must be a database configured for use in the DB2 client, such as a database listed in the Client Configuration Assistant. Also, the database user must be able to select from the table specified in TestTableName.

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

```
<JDBCConnectionPool DriverName="COM.ibm.db2.jdbc.app.DB2Driver"
    Name="MyJDBC Connection Pool"
    Password="{3DES}Pd8QwSJ5FtLEfuiA/vcy3g=="
    Properties="user=dbuser;DatabaseName=db1"
    Targets="myserver"
    TestConnectionsOnCreate="true"
    TestConnectionsOnReserve="true"
    TestTableName="SYSIBM.SYSTABLES"
    URL="jdbc:db2:db1"/>
```

Table 5-2 XA Connection Pool Attributes Using the DB2 Type 2 JDBC Driver

Attribute	Value
URL	`jdbc:db2:`*`dbname`*
Driver Class Name	`COM.ibm.db2.jdbc.DB2XADataSource`
Properties	`user=`*`username`* `DatabaseName=`*`dbname`*
Password	*`password`*
TestConnectionsOnCreate	`true`
TestConnectionsOnReserve	`true`
TestTableName	`SYSIBM.SYSTABLES`
KeepXAConnTillTxComplete	`true`
Target	`serverName`

The database name in the URL and in the Properties string must be a database configured for use in the DB2 client, such as a database listed in the Client Configuration Assistant.

DB2 requires that all processing for a global transaction occurs on a single database connection, so you must set KeepXAConnTillTxComplete to true.

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

```
<JDBCConnectionPool DriverName="COM.ibm.db2.jdbc.DB2XADataSource"
    KeepXAConnTillTxComplete="true"
    Name="My XA JDBC Connection Pool"
    Password="{3DES}Pd8QwSJ5FtLEfuiA/vcy3g=="
    Properties="user=dbuser;DatabaseName=db1"
    Targets="myserver"
    TestConnectionsOnCreate="true"
    TestConnectionsOnReserve="true"
    TestTableName="SYSIBM.SYSTABLES"
    URL="jdbc:db2:db1"/>
```

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.

The following sections describe how to install and configure the Microsoft SQL Server 2000 Driver for JDBC.

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 attributes in Table 5-3 when creating a connection pool that uses the Microsoft SQL Server Driver for JDBC.

Table 5-3 Connection Pool Attributes Using the Microsoft SQL Server Driver for JDBC

Attribute	Value
URL	`jdbc:microsoft:sqlserver://`*`server_name`*`:`*`port`*
Driver Class Name	`com.microsoft.jdbc.sqlserver.SQLServerDriver`

Table 5-3 Connection Pool Attributes Using the Microsoft SQL Server Driver for JDBC

Attribute	Value
Properties	`user=`*`username`* `DatabaseName=`*`dbname`* `selectMethod=cursor`
Password	*`password`*
Target	`serverName`

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

```
<JDBCConnectionPool
    Name="mssqlPool"
    DriverName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    URL="jdbc:microsoft:sqlserver://db4:1433"
    Properties="databasename=db4;user=sa;
    selectMethod=cursor"
    Password="{3DES}vlsUYhxlJ/I="
    InitialCapacity="4"
    CapacityIncrement="2"
    MaxCapacity="10"
    Targets="examplesServer"
/>
```

Note: You must add `selectMethod=cursor` to the list of connection properties in order to use connections in a transactional mode. This enables your applications to have multiple concurrent statements open from a given connection, which is required for pooled connections.

Without setting `selectMethod=cursor`, this JDBC driver creates an internal cloned connection for each concurrent statement, each as a different DBMS user. This makes it impossible to concurrently commit transactions and may cause deadlocks.

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-3.ibm.com/software/data/informix/tools/jdbc/`. 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\weblogic81`.

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\ifxjdbc
   x.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 5-4 and Table 5-5 when creating a connection pool that uses the IBM Informix JDBC driver.

Table 5-4 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`

Table 5-4 Non-XA Connection Pool Attributes Using the Informix JDBC Driver

Attribute	Value
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 5-5 XA Connection Pool Attributes Using the Informix JDBC Driver

Attribute	Value
URL	*leave blank*
Driver Class Name	com.informix.jdbcx.IfxXADataSource

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

Attribute	Value
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:

- 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.

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. 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 2-13 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");
```

```
  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;

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

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

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

...
//Close JDBC objects as soon as possible
    stmt.close();
    stmt=null;

    conn.close();
    conn=null;

 }
  catch (Exception e) {
    // a failure occurred
    log message;
  }
finally {
  try {
    ctx.close();
  } catch (Exception e) {
     log message; }
  try {
    if (rs != null) rs.close();
  } catch (Exception e) {
     log message; }
  try {
    if (stmt != null) stmt.close();
```

```
  } catch (Exception e) {
     log message; }
  try {
    if (conn != null) conn.close();
  } catch (Exception e) {
     log message; }
}
```

(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/docs81/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

Note: BEA strongly discourages directly accessing a physical JDBC connection except for when it is absolutely required. See "Limitations for Using a Physical Connection" on page 5-21.

Standard practice is to cast a connection to the generic JDBC connection (a wrapped physical connection) provided by WebLogic Server. This allows the server instance to manage the connection for the connection pool, enable connection pool features, and maintain the quality of connections provided to applications. Occasionally, a DBMS vendor may provide extra non-standard JDBC-related classes that require direct access of the physical connection (the actual vendor JDBC connection). To directly access a physical connection in a connection pool, you must cast the connection using `getVendorConnection` at `http://e-docs.bea.com/wls/docs81/javadocs/weblogic/jdbc/extensions/WLConnection.html`.

The following sections provide information on getting a physical connection:

Opening a 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 5-15, then do one of the following:

- Implicitly pass the physical connection (using `getVendorConnection`) within a method that requires the physical connection.
- Cast the connection as a `WLConnection` and call `getVendorConnection`.

Always limit direct access of physical database connections to vendor-specific calls. For all other situations, use the generic JDBC connection provided by WebLogic Server. Sample code to open a connection for vendor-specific calls is provided in Listing 5-1.

Listing 5-1 Code Sample to Open a Connection for Vendor-specific Calls

```
//Import this additional class and any vendor packages
//you may need.
import weblogic.jdbc.extensions.WLConnection
.
.
.
myJdbcMethod()
{
  // Connections from a connection pool should always be
  // method-level variables, never class or instance methods.
  Connection conn = null;

   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");

     // Always get a pooled connection in a try block where it is
     // used completely and is closed if necessary in the finally
     // block.
     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();
      // do not close vendorConn

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

      // If you have a vendor-specific method that requires the
      // physical connection, it is best not to obtain or retain
      // the physical connection, but simply pass it implicitly
      // where needed, eg:
//vendor.special.methodNeedingConnection(((WLConnection)conn)).getVendorCo
nnection());
```

Closing a Connection

When you are finished with your JDBC work, you should close the logical connection to get it back into the pool. When you are done with the physical connection:

- Close any objects you have obtained from the connection.
- Do not close the physical connection. Set the physical connection to null.

You determine how a connection closes by setting the value of the `Remove Infected Connections Enabled` property in the administration console. See JDBC Connection Pool --> Configuration --> Connections in the *Administration Console Help*. Sample code to close a vendor-specific connection is shown in Listing 5-2.

Note: The `Remove Infected Connections Enabled` property applies only to applications that explicitly call `getVendorConnection`.

Listing 5-2 Sample Code to Close a Connection for Vendor-specific Calls

```
 // As soon as you are finished with vendor-specific calls,
      // nullify the reference to the connection.
```

```
      // Do not keep it or close it.
      // Never use the vendor connection for generic JDBC.
      // Use the logical (pooled) connection for standard JDBC.
      vendorConn = null;

      ... do all the JDBC needed for the whole method...

      // close the logical (pooled) connection to return it to
      // the connection pool, and nullify the reference.
      conn.close();
      conn = null;
    }

    catch (Exception e)
    {
      // Handle the exception.
    }
    finally
{
      // For safety, check whether the logical (pooled) connection
      // was closed.
      // Always close the logical (pooled) connection as the
      // first step in the finally block.
      if (conn != null) try {conn.close();} catch (Exception ignore){}
    }
}
```

Remove Infected Connections Enabled is True

When `Remove infected Connections Enabled=false` (default value) and you close the logical connection, the server instance discards the underlying physical connection and creates a new connection to replace it. This action ensures that the pool can guarantee to the next user that they are the sole user of the pool connection. This configuration provides a simple and safe way to close a connection. However, there is a performance loss because:

- The physical connection is 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.

Remove Infected Connections Enabled is False

Note: Use `Remove infected Connections Enabled=false` only if you are sure that the exposed physical connection will never be retained or reused after the logical connection is closed.

When `Remove infected Connections Enabled=false` and you close the logical connection, the server instance simply returns the physical connection to the connection pool for reuse. Although this configuration minimizes performance losses, the server instance does not guarantee the quality of the connection or to effectively manage the connection after the logical connection is closed. You must make sure that the connection is suitable for reuse by other applications before it is returned to the connection pool.

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, such as error handling and statement caching.
- You should use the physical connection only for the vendor-specific methods or classes that require it. Do not use the physical connection for generic JDBC, such as creating statements or transactional calls.

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 all 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 submit 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 5-25.

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 handles de-serialization for these object types so they can be used 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 5-41.

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 5-15.

```
// 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 5-21. 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 5-42. 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 5-42. 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 remote 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 5-41.

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 5-26.
- STRUCTS—See "Programming with STRUCTs" on page 5-29.
- REFs—See "Programming with REFs" on page 5-33.
- BLOBs and CLOBs—See "Programming with BLOBs and CLOBs" on page 5-38.

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 remote 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 5-15.

```
// 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 Follow these steps:

1. Create an array in the database using PL/SQL, if the array you want to update does not already exist in the database.
2. Get the ARRAY using a result set or a callable statement.
3. Work with the array in your Java application as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
4. Update the array in the database using 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 5-27.)
2. Get a connection. (See "Establish the Connection" on page 5-27.)
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 5-27.)
2. Get a database connection. (See "Establish the Connection" on page 5-27.)
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 5-42.

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.

Programming with Oracle Virtual Private Databases

An Oracle Virtual Private Database (VPD) is an aggregation of server-enforced, application-defined fine-grained access control, combined with a secure application context in the Oracle 9i database server. To use VPDs in your WebLogic Server application, you would typically do the following:

1. Create a JDBC connection pool in your WebLogic Server configuration that uses either the Oracle Thin driver or the Oracle OCI driver. See "Configuring and Using WebLogic JDBC" on page 2-1 or "Creating and Configuring a JDBC Connection Pool" in the *Administration Console Online Help*.

   Note: If you are using an XA-enabled version of the JDBC driver, you must set `KeepXAConnTillTxComplete=true`. See "Additional XA Connection Pool Properties" in the *Administration Console Online Help*.

   The WebLogic jDriver for Oracle cannot propagate the `ClientIdentifier`, so it is ineffective to use the driver with VPDs.

2. Create a data source in your WebLogic Server configuration that points to the connection pool. See "Configuring and Using DataSources" on page 2-13 or "Creating and Configuring a JDBC Data Source" in the *Administration Console Online Help*.

3. Do the following in your application:

```
import weblogic.jdbc.extensions.WLConnection

// get a connection from a WLS JDBC connection pool
Connection conn = ds.getConnection();

// Get the underlying vendor connection object
oracle.jdbc.OracleConnection orConn = (oracle.jdbc.OracleConnection)
   (((WLConnection)conn).getVendorConnection());

// Set CLIENT_IDENTIFIER (which will be accessible from
// USERENV naming context on the database server side)
orConn.setClientIdentifier(clientId);

/* perform application specific work, preferably using conn instead of
orConn */

// clean up connection before returning to WLS JDBC connection pool
orConn.clearClientIdentifier(clientId);
```

```
// As soon as you are finished with vendor-specific calls,
// nullify the reference to the physical connection.
orConn = null;

// close the pooled connection
conn.close();
```

Note: This code uses an underlying physical connection from a pooled (logical) connection. See "Getting a Physical Connection from a Connection Pool" on page 5-17 for usage guidelines.

Oracle VPD with WebLogic Server 8.1SP2

Starting with WebLogic Server 8.1 SP2, WebLogic Server provides support for the `oracle.jdbc.OracleConnection.setClientIdentitfier` and `oracle.jdbc.OracleConnection.clearClientIndentifier` methods without using the underlying physical connection from a pooled connection. To use VPDs in your WebLogic Server application, you would typically do the following:

```
import weblogic.jdbc.vendor.oracle.OracleConnection;

// get a connection from a WLS JDBC connection pool
Connection conn = ds.getConnection();

// cast to the Oracle extension and set CLIENT_IDENTIFIER
// (which will be accessible from USERENV naming context on
//  the database server side)
((weblogic.jdbc.vendor.oracle.OracleConnection)conn).setClientIdentifier(c
lientId);

/* perform application specific work */

// clean up connection before returning to WLS JDBC connection pool
((OracleConnection)conn).clearClientIdentifier(clientId);

// close the connection
conn.close();
```

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`
- `weblogic.jdbc.vendor.oracle.OracleThinBlob`
- `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 5-21 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 5-6. BEA encourages you to use the alternative interface listed in the table.

Table 5-6 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`
`weblogic.jdbc.vendor.oracle.OraclePreparedStatement`	`oracle.jdbc.OraclePreparedStatement`
`weblogic.jdbc.vendor.oracle.OracleResultSet`	`oracle.jdbc.OracleResultSet`

The interfaces listed in Table 5-7 are still valid because Oracle does not provide interfaces to access these extension methods.

Table 5-7 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 5-8 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (This interface is **deprecated**. See Table 5-6.)	`void clearClientIdentifier(String s) throws java.sql.SQLException;`
	`boolean getAutoClose() 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;`

Table 5-8 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-6.)	`boolean getRestrictGetTables()`
	`  throws java.sql.SQLException;`
	`boolean getUsingXAFlag()`
	`  throws java.sql.SQLException;`
	`boolean getXAErrorFlag()`
	`  throws java.sql.SQLException;`
	`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;`

Table 5-8 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-6.)	`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;`
	`void removeAllDescriptor() throws java.sql.SQLException;`
	`void removeDescriptor(String s) throws java.sql.SQLException;`
	`void setAutoClose(boolean on) throws java.sql.SQLException;`
	`void setClientIdentifier(String s) throws java.sql.SQLException;`
	`void clearClientIdentifier(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;`

Table 5-8 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-6.)	`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 5-9 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement`
(This interface is **deprecated**. See Table 5-6.)
	`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;`

Table 5-9 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement` (continued) (This interface is **deprecated**. See Table 5-6.)	`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;`
	`void defineColumnType(int i, int j, int k)`
	`  throws java.sql.SQLException;`
	`void describe()`
	`  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 5-10 OracleResultSet Interface

Extends	Method Signature

`OracleResultSet` extends `java.sql.ResultSet`
(This interface is **deprecated**. See Table 5-6.)
	`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 5-11 OracleCallableStatement Interface

Extends	Method Signature

`OracleCallableStatement` extends `java.sql.CallableStatement`
(This interface is **deprecated**. See Table 5-6.)
	`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 5-12 OraclePreparedStatement Interface

Extends	Method Signature

`OraclePreparedStatement` extends `OracleStatement` and `java.sql.PreparedStatement`
(This interface is **deprecated**. See Table 5-6.)
	`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 5-13 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 5-13 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 5-14 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 5-15 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 5-16 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 5-17 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

Using RowSets with WebLogic Server

This section includes the following information about using JDBC rowSets with WebLogic Server:

About RowSets

RowSets are a JDBC 2.0 extension to the `java.sql.ResultSet` interface. The WebLogic Server implementation of RowSets provides a disconnected RowSet. In this model, a RowSet object is populated from the database and then the database cursor and connection are immediately released. The RowSet is disconnected from the database and provides a ResultSet

interface for the cached data. The user may read, modify, delete, or even insert new rows into the RowSet in memory. When `acceptChanges` is called, the RowSet takes all of the in-memory updates and writes them back to the database.

In most cases, populating a RowSet data and updating the database occur in separate transactions. The RowSet implementation uses optimistic concurrency control to ensure data consistency.

WebLogic Server RowSets implementation implements and extends `java.io.Serializable`, so the RowSet can be sent as an RMI parameter or return value. For example, an EJB method could populate a RowSet from a database query and then return the RowSet to a client.

RowSets can also read and write their state and metadata to an XML format. The RowSet metadata is written as an XML schema document and the RowSet data is written as an XML document that conforms to the schema. You can also populate the metadata and cached data for a RowSet from XML documents.

Note: When using a RowSet in a client-side application, the *exact same* JDBC driver classes must be in the `CLASSPATH` on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

Creating RowSets

RowSets are created from a factory interface.

```
import weblogic.jdbc.rowset.RowSetFactory;
import weblogic.jdbc.rowset.WLCachedRowSet;

RowSetFactory factory = RowSetFactory.newInstance();
WLCachedRowSet rowSet = factory.newCachedRowSet();
```

Working with Data in a RowSet

The following sections describe how to populate a RowSet, manipulate the data in the RowSet, and then flush the changes to the database.

Note: Delimiter identifiers may not be used for column or table names in RowSets. Delimiter identifiers are identifiers that need to be enclosed in double quotation marks when appearing in a SQL statement. They include identifiers that are SQL reserved words (e.g., `USER`, `DATE`, etc.) and names that are not identifiers. A valid identifier must start with a letter and contain only letters, numbers, and underscores.

Populating a RowSet

After the RowSet object is created, its cache can be filled with data. Once a RowSet has been populated, it is disconnected from the database and acts as a memory cache. There are three methods sources for populating the RowSet's cache with data:

- An existing result set. See "Populating a RowSet from an Existing ResultSet" on page 6-3.
- A Database Query. See "Populating a RowSet from a DataSource and Query" on page 6-3.
- An XML Document. See "Populating a RowSet from an XML Document" on page 6-16.

Populating a RowSet from an Existing ResultSet

A RowSet can be populated from an existing JDBC ResultSet. This is a common case when data is read from a stored procedure or JDBC code already exists to load the data. The RowSet can be loaded by calling its populate method.

```
rowSet.populate(myResultSet);
```

Populating a RowSet from a DataSource and Query

A RowSet can be populated by providing database connection information and a SQL query. First, you provide the CachedRowSet with information needed to get a JDBC connection. This can be done by providing a `javax.sql.DataSource` object, a DataSource JNDI name, or a JDBC Driver URL. The DataSource API is recommended since it is the standard JDBC 2.0 method for retrieving JDBC connections. Also, only connections retrieved via the DataSource API can participate in XA/2PC transactions.

```
rowSet.setDataSourceName("myDataSource");
```

If necessary, the `setUsername` and `setPassword` methods can be used to set the credentials necessary to access your DataSource in WebLogic Server.

```
rowSet.setUsername("weblogic");
rowSet.setPassword("weblogic");
```

Next, specify a SQL query to use to load the database. For instance, the following query populates the RowSet with all employees with a salary greater than 50000:

```
rowSet.setCommand("select e_name, e_id from employees WHERE e_salary > ?");
rowSet.setInt(1, 50000);
```

Finally, run the execute method that runs the specified query and loads the RowSet with data. The execute method closes the JDBC connection. The RowSet does not maintain open cursors or connections to the database.

```
rowSet.execute();
```

Retrieving Data from a RowSet

Because the RowSet is an extension to the ResultSet interfaces, it inherits all of the ResultSet methods for retrieving data. As with a ResultSet, you can iterate through a RowSet using the `next()` method. The `getXXX` methods can be used to read data from the RowSet.

```
while(rowSet.next()) {
String name = rowSet.getString("e_name");
int id      = rowSet.getInt("e_id");

System.out.println("Read name: "+name+ " id: "+id);
}

while(rowSet.next()) {
  String name = rowSet.getString("e_name");
  int id      = rowSet.getInt("e_id");

  System.out.println("Read name: "+name+ " id: "+id);
}
```

Updating Data in a RowSet

RowSets use the ResultSet `updateXXX` methods for updating data.

It is important to understand that RowSet updates are kept in memory only. Updates are written back to the database only when you call the `acceptChanges` method.

```
// move back to the beginning of the rowSet
rowSet.beforeFirst();
while(rowSet.next()) {
  String name = rowSet.getString("e_name");
  // convert to upper case
  name = name.toUpper();
  rowSet.updateString("e_name", name);
  rowSet.updateRow();
}
```

```
// Call acceptChanges to write all of the in-memory updates to the database

rowSet.acceptChanges();
```

Note: You must call `RowSet.updateRow` or `RowSet.cancelRowUpdates` before moving the RowSet's cursor with the `next` method.

Deleting Data from a RowSet

Deleting rows is very similar to updating rows. The `deleteRow()` method marks a row for deletion. When you call the `acceptChanges` method, the RowSet issues the appropriate SQL to delete the selected rows.

```
// move back to the beginning of the rowSet
rowSet.beforeFirst();

while(rowSet.next()) {

  String name = rowSet.getString("e_name");

  if ("Rob".equals(name)) {
    rowSet.deleteRow();
  }
}

// When acceptChanges all of the in-memory deletions are written to
// the database

rowSet.acceptChanges();
```

Inserting Data into a RowSet

Like ResultSets, RowSets have the concept of a special insert row. To insert data, call `moveToInsertRow` and then update the values in the row. The `insertRow` method is called to indicate that the updates are done. You can either insert another row, or call `moveToCurrentRow` to return to the read data. When you call `acceptChanges`, all inserted rows are sent to the database.

```
rowSet.moveToInsertRow();

rowSet.updateString("e_name", "Seth");
rowSet.updateInt("e_id", 2);
rowSet.insertRow();
```

```
rowSet.updateString("e_name", "Matt");
rowSet.updateInt("e_id", 3);
rowSet.insertRow();

rowSet.moveToCurrentRow();

// issues SQL INSERTs to database
rowSet.acceptChanges();
```

Flushing Changes to the Database

A RowSet acts like a database cache, and all updates to it occur in memory. To flush these changes back to the database, call the `acceptChanges` method.

The RowSet's `acceptChanges` method uses the DataSource or connection information to acquire a database connection. It then issues all of the INSERT, UPDATE, or DELETE statements that have been made in memory to the database.

Since the RowSet was disconnected from the database and not holding any locks or database resources, it is possible that the underlying data in the database has been changed since the RowSet was populated. The RowSet implementation uses optimistic concurrency control on its UPDATE and DELETE statements to check for stale data. See "Optimistic Concurrency Policies" on page 6-6 for details.

RowSet Meta Data

The RowSet API provides a `getMetaData` method for access to the associated `javax.sql.RowSetMetaData` object. The WLCachedRowSet implementation provides a WLRowSetMetaData interface that extends the standard RowSetMetaData with additional functionality.

The metadata can be accessed with:

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();
```

Optimistic Concurrency Policies

In most cases, populating a RowSet with data and updating the database occur in separate transactions. The underlying data in the database can change in the time between the two transactions. The WebLogic Server RowSet implementation uses optimistic concurrency control to ensure data consistency.

With optimistic concurrency, RowSets work on the assumption that multiple users are unlikely to change the same data at the same time. Therefore, as part of the disconnected RowSet model, the RowSet does not lock database resources. However, before writing changes to the database, the RowSet must check to make sure that the data to be changed in the database has not already changed since the data was read into the RowSet.

The UPDATE and DELETE statements issued by the RowSet include WHERE clauses that are used to verify the data in the database against what was read when the RowSet was populated. If the RowSet detects that the underlying data in the database has changed, it issues an `OptimisticConflictException`. The application can catch this exception and determine how to proceed. Typically, applications will refresh the updated data and present it to the user again.

The WLCachedRowSet implementation offers several optimistic concurrency policies that determine what SQL the RowSet issues to verify the underlying database data:

- VERIFY_READ_COLUMNS
- VERIFY_MODIFIED_COLUMNS
- VERIFY_SELECTED_COLUMNS
- VERIFY_NONE
- VERIFY_AUTO_VERSION_COLUMNS
- VERIFY_VERSION_COLUMNS

To illustrate the differences between these policies, we will use an example that uses the following:

- A very simple employees table with 3 columns:

```
CREATE TABLE employees (
  e_id integer primary key,
  e_salary integer,
  e_name varchar(25)
);
```

- A single row in the table:

```
e_id = 1, e_salary = 10000, and e_name = 'John Smith'
```

In the example for each of the optimistic concurrency policies listed below, the RowSet will read this row from the employees table and set John Smith's salary to 20000. The example will then show how the optimistic concurrency policy affects the SQL code issued by the RowSet.

VERIFY_READ_COLUMNS

The default RowSet optimistic concurrency control policy is VERIFY_READ_COLUMNS. When the RowSet issues an UPDATE or DELETE, it includes all columns that were read from the database in the WHERE clause. This verifies that the value in all columns that were initially read into the RowSet have not changed.

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000 AND e_name = 'John Smith';
```

VERIFY_MODIFIED_COLUMNS

The VERIFY_MODIFIED_COLUMNS policy only includes the primary key columns and the updated columns in the WHERE clause. It is useful if your application only cares if its updated columns are consistent. It does allow your update to commit if columns that have not been updated have changed since the data has been read.

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000
```

The `e_id` column is included since it is a primary key column. The `e_salary` column is a modified column so it is included as well. The `e_name` column was only read so it is not verified.

VERIFY_SELECTED_COLUMNS

The VERIFY_SELECTED_COLUMNS includes the primary key columns and columns you specify in the WHERE clause.

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();
metaData.setOptimisticPolicy(WLRowSetMetaData.VERIFY_SELECTED_COLUMNS);
// Only verify the e_salary column
metaData.setVerifySelectedColumn("e_salary", true);

metaData.acceptChanges();
```

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000
```

The e_id column is included since it is a primary key column. The e_salary column is a selected column so it is included as well.

VERIFY_NONE

The VERIFY_NONE policy only includes the primary key columns in the WHERE clause. It does not provide any additional verification on the database data.

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000 WHERE e_id = 1
```

VERIFY_AUTO_VERSION_COLUMNS

The VERIFY_AUTO_VERSION_COLUMNS includes the primary key columns as well as a separate version column that you specify in the WHERE clause. The RowSet will also automatically increment the version column as part of the update. This version column must be an integer type. The database schema must be updated to include a separate version column (e_version). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.
  VERIFY_AUTO_VERSION_COLUMNS);

metaData.setAutoVersionColumn("e_version", true);

metaData.acceptChanges();
```

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000, e_version = 2
WHERE e_id = 1 AND e_version = 1
```

The e_version column is automatically incremented in the SET clause. The WHERE clause verified the primary key column and the version column.

VERIFY_VERSION_COLUMNS

The VERIFY_VERSION_COLUMNS has the RowSet check the primary key columns as well as a separate version column. The RowSet does not increment the version column as part of the update. The database schema must be updated to include a separate version column (e_version). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.VERIFY_VERSION_COLUMNS);

metaData.setVersionColumn("e_version", true);
```

```
metaData.acceptChanges();
```

In our example update, the RowSet issues:

```
UPDATE employees SET e_salary = 20000
WHERE e_id = 1 AND e_version = 1
```

The WHERE clause verifies the primary key column and the version column. The RowSet does not increment the version column so this must be handled by the database. Some databases provide automatic version columns that increment when the row is updated. It is also possible to use a database trigger to handle this type of update.

Optimistic Concurrency Control Limitations

The Optimistic policies only verify UPDATE and DELETE statements against the row they are changing. Read-only rows are not verified against the database.

Most databases do not allow BLOB or CLOB columns in the WHERE clause so the RowSet never verifies BLOB or CLOB columns.

When multiple tables are included in the RowSet, the RowSet only verifies tables that have been updated.

Choosing an Optimistic Policy

The default VERIFY_READ_COLUMNS provides a strong-level of consistency at the expense of some performance. Since all columns that were initially read must be sent to the database and compared in the database, there is some additional overhead to this policy. VERIFY_READ_COLUMNS is appropriate when strong levels of consistency are needed, and the database tables cannot be modified to include a version column.

The VERIFY_SELECTED_COLUMNS is useful when the developer needs complete control over the verification and wants to use application-specific knowledge to fine-tune the SQL.

The VERIFY_AUTO_VERSION_COLUMNS provides the same level of consistency as VERIFY_READ_COLUMNS but only has to compare a single integer column. This policy also handles incrementing the version column so it requires a minimal amount of database setup.

The VERIFY_VERSION_COLUMNS is recommended for production systems that want the highest level of performance and consistency. Like VERIFY_AUTO_VERSION_COLUMNS, it provides a high level of consistency while only incurring a single column comparison in the database. VERIFY_VERSION_COLUMNS requires that the database handle incrementing the

version column. Some databases provide a column type that automatically increments itself on updates, but this behavior can also be implemented with a database trigger.

The VERIFY_MODIFIED_COLUMNS and VERIFY_NONE decrease the consistency guarantees, but they also decrease the likelihood of an optimistic conflict. You should consider these policies when performance and avoiding conflicts outweigh the need for higher level of data consistency.

MetaData Settings for RowSet Updates

When data is read into a RowSet, the RowSet implementation uses the ResultSetMetaData interface to automatically learn the table and column names of the read data. In many cases, this is enough information for the RowSet to generate the required SQL for writing changes back to the database. However, many JDBC drivers just return an empty string when asked for the table name of a given column. Without the table name, the RowSet can be used for read-only operations only. The RowSet cannot issue updates unless the table name is specified programmatically.

The RowSet implementation provides an extended MetaData interface that allows you to specify schema information that cannot be automatically determined via the JDBC Driver. The WLRowSetMetaData interface can be used to set the schema information.

executeAndGuessTableName and executeAndGuessTableNameAndPrimaryKeys

When a RowSet is populated via a SQL query, the `execute()` method is generally used to run the query and read the data. The WLCachedRowSet implementation provides the `executeAndGuessTableName` and `executeAndGuessTableNameAndPrimaryKeys` methods that extend the execute method to also determine the associated table metadata.

The `executeAndGuessTableName` method parses the associated SQL and sets the table name for all columns as the first word following the SQL keyword `FROM`.

The `executeAndGuessTableNameAndPrimaryKeys` method parses the SQL command to read the table name. It then uses the `java.sql.DatabaseMetaData` to determine the table's primary keys.

Setting Table and Primary Key Information Using the MetaData Interface

You can also choose to set the table and primary key information using the RowSetMetaData interface.

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();

// convenience method to set one table name for all columns
metaData.setTableName("employees");
```

or

```
metaData.setTableName("e_id", "employees");
metaData.setTableName("e_name", "employees");
```

You can also use the WLRowSetMetaData to identify primary key columns.

```
metaData.setPrimaryKeyColumn("e_id", true);
```

Setting the Write Table

The WLRowSetMetaData interface includes the `setWriteTableName` method to indicate the only table that should be updated or deleted. This is typically used when a RowSet is populated via a join from multiple tables, but the RowSet should only update one table. Any column that is not from the write table is marked as read-only.

For instance, a RowSet might include a join of orders and customers. The write table could be set to orders. If deleteRow were called, it would delete the order row, but not delete the customer row.

RowSets and Transactions

Most database or JDBC applications use transactions, and RowSets support transactions, including JTA transactions. The common use case is to populate the RowSet in Transaction 1. Transaction 1 commits, and there are no database or application server locks on the underlying data. The RowSet holds the data in-memory, and it can be modified or shipped over the network to a client. When the application wishes to commit the changes to the database, it starts Transaction 2 and calls the RowSet's `acceptChanges` method. It then commits Transaction 2.

Integrating with JTA Global Transactions

The EJB container and the UserTransaction interface start transactions with the JTA transaction manager. The RowSet operations can participate in this transaction. To participate in the JTA transaction, the RowSet *must* use a transaction-aware DataSource (TxDataSource). The DataSource can be configured in the WebLogic Server console.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet aborts the global JTA transaction. The application will typically re-read the data and process the update again in a new transaction.

Behavior of Rowsets Using Global Transactions

In the case of a failure or rollback, the data is rolled back from the database, but is not rolled back from the rowset. Before proceeding you should do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.
- Create a new rowset with current data.

Using Local Transactions

If a JTA global transaction is not being used, the RowSet uses a local transaction. It first calls `setAutoCommit(false)` on the connection, then it issues all of the SQL statements, and finally it calls `connection.commit()`. This attempts to commit the local transaction. This method should *not* be used when trying to integrate with a JTA transaction that was started by the EJB or JMS containers.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet rolls back the local transaction. In this case, none of the SQL issued in `acceptChanges` will commit to the database.

Behavior of Rowsets Using Local Transactions

This section provides information on the behavior of rowsets in failed local transactions. The behavior depends on the type of connection object:

Calling connection.commit

In this situation, the connection object is not created by the rowset and initiates a local transaction by calling `connection.commit`. If the transaction fails or if the connection calls

`connection.rollback`, the data is rolled back from the database, but is not rolled back in the rowset. Before proceeding, you must do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.
- Create a new rowset with current data.

Calling acceptChanges

In this situation, the rowset creates its own connection object and uses it to update the data in rowset by calling `acceptChanges`. In the case of failure or if the rowset calls `connection.rollback`, the data is be rolled back from the rowset and also from the database.

Performance Options

Consider the following performance options when using RowSets.

JDBC Batching

The RowSet implementation includes support for JDBC 2.0 batch operations. Instead of sending each SQL statement individually to the JDBC driver, a batch sends a collection of statements in one bulk operation to the JDBC driver. Batching is disabled by default, but it generally improves performance when large numbers of updates occur in a single transaction. It is worthwhile to benchmark with this option enabled and disabled for your application and database.

The WLCachedRowSet interface contains the methods `setBatchInserts(boolean)`, `setBatchDeletes(boolean)`, and `setBatchUpdates(boolean)` to control batching of INSERT, DELETE, and UPDATE statements.

Note: The `setBatchInserts`, `setBatchDeletes`, or `setBatchUpdates` methods must be called before the `acceptChanges` method is called.

Oracle Batching Limitations

Since the WLCachedRowSet relies on optimistic concurrency control, it needs to determine whether an update or delete command has succeeded or an optimistic conflict occurred. The WLCachedRowSet implementation relies on the JDBC driver to report the number of rows updated by a statement to determine whether a conflict occurred or not. In the case where 0 rows were updated, the WLCachedRowSet knows that a conflict did occur.

Oracle JDBC drivers return `java.sql.Statement.SUCCESS_NO_INFO` when batch updates are executed, so the RowSet implementation cannot use the return value to determine whether a conflict occurred.

When the RowSet detects that batching is used with an Oracle database, it automatically changes its batching behavior:

Batched inserts perform as usual since they are not verified.

Batched updates run as normal, but the RowSet issues an extra SELECT query to check whether the batched update encountered an optimistic conflict.

Batched deletes use group deletes since this is more efficient than executing a batched delete followed by a SELECT verification query.

Group Deletes

When multiple rows are deleted, the RowSet would normally issue a DELETE statement for each deleted row. When group deletes are enabled, the RowSet issues a single DELETE statement with a WHERE clause that includes the deleted rows.

For instance, if we were deleting 3 employees from our table, the RowSet would normally issue:

```
DELETE FROM employees WHERE e_id = 3 AND e_version = 1;
DELETE FROM employees WHERE e_id = 4 AND e_version = 3;
DELETE FROM employees WHERE e_id = 5 AND e_version = 10;
```

When group deletes are enabled, the RowSet issues:

```
DELETE FROM employees
WHERE e_id = 3 AND e_version = 1 OR
      e_id = 4 AND e_version = 3 OR
      e_id = 5 AND e_version = 10;
```

The programmer can use the `WLRowSetMetaData.setGroupDeleteSize` to determine the number of rows included in a single DELETE statement. The default value is 50.

RowSets and XML

The WLCachedRowSet implementation provides support for writing its metadata as an XML schema document and its data as an XML document that conforms to the schema. The WLCachedRowSet can also populate itself and its metadata from an existing XML schema and XML document.

For instance, a RowSet can be converted to XML and sent as an XML message to another process. The other process could rebuild the RowSet instance in memory, read and update data, and send the response back as another XML message. Finally the original server could convert the XML message back to a RowSet and update the database.

Writing a RowSet Instance as XML

The WLRowSetMetaData interface contains the method `writeXMLSchema` to write the RowSetMetaData as an XML schema document. The WLRowSetMetaData interface has a writeXML method for converting the RowSet's data into an XML instance document.

```
XMLOutputStreamFactory xoFactory =
  XMLOutputStreamFactory.newInstance();

WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();

XMLOutputStream xos = null;

// Write XSD Schema
try {
  xos = xoFactory.newDebugOutputStream(new
    FileOutputStream("rowset.xsd");
  metaData.writeXMLSchema(xos);
} finally {
  if (xos != null) xos.close();
}

// Write XML Instance data
try {
  xos = xoFactory.newDebugOutputStream(new
    FileOutputStream("rowset.xml");
  rowSet.writeXML(xos);
} finally {
  if (xos != null) xos.close();
}
```

Populating a RowSet from an XML Document

The WLRowSetMetaData interface contains the method `loadXMLSchema` to load the RowSetMetaData from an XML schema document. The WLRowSetMetaData interface has a `loadXML` method for populating from an XML instance document.

```
XMLInputStreamFactory xiFactory =
  XMLInputStreamFactory.newInstance();
XMLInputStream xis = null;
WLCachedRowSet rowSet = factory.newCachedRowSet();
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();
// Read XSD
try {
  xis = xiFactory.newInputStream(new FileInputStream("rowset.xsd"));
        metaData.loadXMLSchema(xis);
} finally {
  if (xis != null) xis.close();
}
// Read XML
try {
  xis = xiFactory.newInputStream(new FileInputStream("rowset.xml"));
  rs.loadXML(xis);
} finally {
  if (xis != null) xis.close();
}
```

JDBC Type to XML Schema Type Mapping

Table 6-1 JDBC Type to XML Schema Type Mapping

JDBC Type	XML Schema Type
BIGINT	xsd:long
BINARY	xsd:base64Binary
BIT	xsd:boolean
BLOB	xsd:base64Binary
BOOLEAN	xsd:boolean
CHAR	xsd:string
DATE	xsd:dateTime

Table 6-1 JDBC Type to XML Schema Type Mapping

JDBC Type	XML Schema Type
DECIMAL	xsd:decimal
DOUBLE	xsd:decimal
FLOAT	xsd:float
INTEGER	xsd:int
LONGVARBINARY	xsd:base64Binary
LONGVARCHAR	xsd:string
NUMERIC	xsd:integer
REAL	xsd:double
SMALLINT	xsd:short
TIME	xsd:dateTime
TIMESTAMP	xsd:dateTime
TINYINT	xsd:byte
VARBINARY	xsd:base64Binary
VARCHAR	xsd:string

XML Schema Type to JDBC Type Mapping

Table 6-2 XML Schema Type to JDBC Type Mapping

XML Schema Type	JDBC Type
base64Binary	BINARY
boolean	BOOLEAN
byte	SMALLINT
dateTime	DATE
decimal	DECIMAL

Table 6-2 XML Schema Type to JDBC Type Mapping

XML Schema Type	JDBC Type
double	DOUBLE
float	FLOAT
hexBinary	BINARY
int	INTEGER
integer	NUMERIC
long	BIGINT
short	SMALLINT
string	VARCHAR

Multi-table RowSet Mapping

RowSets can be populated from SQL queries that return columns from multiple tables. It is important to understand the RowSet semantics when dealing with multiple tables and the SQL issued by RowSets in multi-table scenarios.

The RowSet optimistic concurrency control policies only verify tables that have been updated. If a RowSet is populated with columns from tables `t1` and `t2`, and column `C` from table `t1` is updated, there will be no SQL UPDATE or SELECT that verifies the values read from table `t2`.

RowSets do not recognize foreign key or other constraints between tables, so when updating multiple tables, it is possible that RowSet updates will fail because of integrity constraints between tables.

Multi-table RowSets work well when a RowSet is a join from N tables with a single write table. For instance, a query might join in several tables but only update the employees table. In this case, the programmer should call `setWriteTableName` to ensure that updates and deletes only apply to the write table.

Another common multi-table scenario is multiple tables that share the same primary key space. This is one logical table that has been split over multiple physical tables in the database. In this scenario, the RowSet will be able to update multiple tables.

Since multi-table RowSets can have complicated update semantics, it is recommended that users set the write table name and only update a single table.

Multi-Table RowSet Example

Consider a simple order entry system that has customer and order tables.

```
CREATE TABLE customer (
  id integer primary key,
  name varchar(200),
  email varchar(200)
);

CREATE TABLE order (
  id integer primary key,
  sku integer,
  quantity integer,
  customer_id integer,
  foreign key customer_id references customer(id)
);
```

This example shows a 1 to many relationship where each customer may have many orders.

A customer portal application might issue a query that loads a customer's current orders and some information about the customer with SQL like this:

```
SELECT o.id, o.sku, o.quantity, c.name, c.email
       FROM order o, customer c
       WHERE c.id = o.customer_id
```

This data will be read into the RowSet with one row containing the matching order and customer columns from the SQL Join.

In cases like this, it is recommended that the many side (order) be set as the write table. This ensures that the one side (customer) is read-only. This allows the user to update details in their order, but will prevent changes to their customer record. This is especially useful for deletes since calling `deleteRow` will delete the order record but will not delete the customer.

CHAPTER 7

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/docs81/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/docs81/ConsoleHelp/jdbc_connection_pools.html`.

For information about using MBeans, see Programming WebLogic JMX Services at `http://e-docs.bea.com/wls/docs81/jmx/index.html`.

Validating a DBMS Connection from the Command Line

Use the `utils.dbping` BEA utility to test two-tier JDBC database connections after you install WebLogic Server. 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.

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) {

  // handle any exceptions as appropriate

}

finally {

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

}
```

Abandoning JDBC Objects

You should also avoid the following practice, which creates abandoned JDBC objects:

```
//Do not do this.
stmt.executeQuery();
rs = stmt.getResultSet();

//Do this instead
rs = stmt.executeQuery();
```

The first line in this example creates a result set that is lost and can be garbage collected immediately.

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, `ldd` 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_PATH
```

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/docs81/oracle/install_jdbc.html.

Using Microsoft SQL with Nested Triggers

The following section provides troubleshooting information when using nested triggers on some Microsoft SQL databases:

For information on supported data bases and data base drivers, see Supported Configurations.

Exceeding the Nesting Level

You may encounter a SQL Server error indicating that the nesting level has been exceeded on some SQL Server databases.

For example:

```
CREATE TABLE EmployeeEJBTable (name varchar(50) not null,salary int, card
varchar(50), primary key (name))


CREATE TABLE CardEJBTable (cardno varchar(50) not null, employee
varchar(50), primary key (cardno), foreign key (employee) references
EmployeeEJB Table(name) on delete cascade)

CREATE TRIGGER  card on EmployeeEJBTable for delete as delete
CardEJBTable  where employee in (select name from deleted)


CREATE TRIGGER  emp on CardEJBTable for delete as delete EmployeeEJBTable
where card in (select cardno from deleted)


insert into EmployeeEJBTable values ('1',1000,'1')

insert into CardEJBTable values ('1','1')

DELETE FROM CardEJBTable WHERE cardno = 1
```

Results in the following error message:

```
Maximum stored procedure, function, trigger, or view nesting level
exceeded  (limit 32).
```

To work around this issue, do the following:

1. Run the following script to reset the nested trigger level to 0:

```
-- Start batch
exec sp_configure 'nested triggers', 0  -- This set's the new value.
reconfigure with override -- This makes the change permanent
-- End batch
```

2. Verify the current value the SQL server by running the following script:

```
exec sp_configure 'nested triggers'
```

Using Triggers and EJBs

Applications using EJBs with a Microsoft driver may encounter situations when the return code from the `execute()` method is 0, when the expected value is 1 (1 record deleted).

For example:

```
CREATE TABLE EmployeeEJBTable (name varchar(50) not null,salary int, card
varchar(50), primary key (name))


CREATE TABLE CardEJBTable (cardno varchar(50) not null, employee
varchar(50), primary key (cardno), foreign key (employee) references
EmployeeEJB Table(name) on delete cascade)

CREATE TRIGGER  emp on CardEJBTable for delete as delete EmployeeEJBTable
where card in (select cardno from deleted)


insert into EmployeeEJBTable values ('1',1000,'1')

insert into CardEJBTable values ('1','1')

DELETE FROM CardEJBTable WHERE cardno = 1
```

The EJB code assumes that the record is not found and throws an appropriate error message.

To work around this issue, run the following script:

```
exec sp_configure 'show advanced options', 1
reconfigure with override
exec sp_configure 'disallow results from triggers',1
reconfigure with override
```

APPENDIX A

Using WebLogic Server with Oracle RAC

More and more customers are looking for solutions to make their back-end systems more scalable and more available. In response to these requests, BEA supports Oracle Real Application Clusters (RAC) for use with WebLogic Server.

The following sections describe the requirements and configuration tasks for using Oracle Real Application Clusters with WebLogic Server:

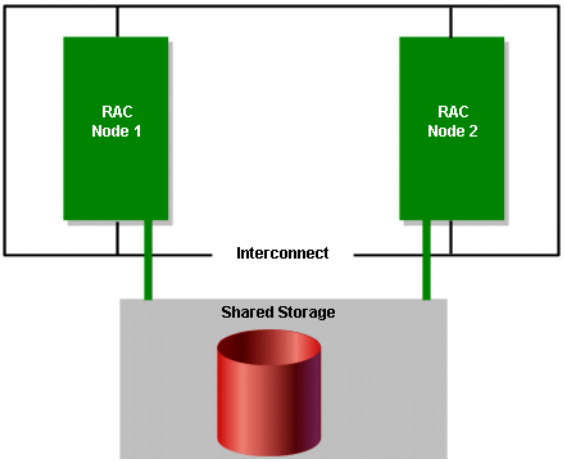
- Overview of Oracle Real Application Clusters
- Environment
- Configuration Considerations for Oracle
- Configuration Options in WebLogic Server with Oracle RAC
- XA Considerations and Limitations with Oracle 9i RAC
- JMS Store Recovery with Oracle RAC

Both Oracle RAC and WebLogic Server are complex systems. To use them together requires specific configuration on both systems, as well as clustering software and a shared storage solution. This document describes the configuration required at a high level. For more details about configuring Oracle RAC, your clustering software, your operating system, and your storage solution, see the documentation from the respective vendors.

Overview of Oracle Real Application Clusters

Oracle Real Application Clusters (RAC) is a software component you can add to a high-availability solution that enables users on multiple machines to access a single database with increased performance. RAC comprises two or more Oracle database instances running on two or more clustered machines and accessing a shared storage device via cluster technology. To support this architecture, the machines that host the database instances are linked by a high-speed interconnect to form the cluster. The interconnect is a physical network used as a means of communication between the nodes of the cluster. Cluster functionality is provided by the operating system or compatible third party clustering software. Figure A-1 shows an Oracle RAC configuration.

Figure A-1 Oracle Real Application Clusters Configuration



Oracle RAC offers features in the following areas:

- Scalability
- Availability
- Load balancing
- Failover

Oracle RAC Scalability with WebLogic Server

An Oracle RAC installation appears like a single standard Oracle database and is maintained using the same tools and practices. All the nodes in the cluster execute transactions against the same database and RAC coordinates each node's access to the shared data to maintain consistency and ensure integrity. You can add nodes to the cluster easily and there is no need to partition data when you add them. This means that you can horizontally scale the database tier as usage and demand grows by adding RAC nodes, storage, or both. You can then scale WebLogic Server by adding a connection pool that maps to the new node.

Oracle RAC Availability with WebLogic Server

Because every RAC node in the cluster has equal access and authority, the loss of a node may impact performance but does not result in downtime. Depending upon your configuration, when a RAC node fails, in-flight transactions are redirected to another node in the cluster either by WebLogic Server or by the Oracle Thin driver. Note that Oracle RAC does not provide failover for database connections; nor does WebLogic Server. But transactions are failed over in the sense that they are heuristically driven to completion, based on the time of the failure.

Oracle RAC Load Balancing with WebLogic Server

If your application requires load balancing across RAC nodes when using global transactions (XA), BEA supports this capability through use of JDBC MultiPools with Oracle RAC nodes. The connection pools that form a Multipool are accessed using a round-robin scheme. When switching connections, WebLogic Server selects a connection from the next connection pool in the order listed. For information on which version(s) of WebLogic Server support JDBC MultiPools, Weblogic Platform Supported Configurations. For more information about using MultiPools with Oracle RAC, see Using MultiPools with Oracle RAC.

In a configuration without a MultiPool, WebLogic Server relies on the connect-time failover feature provided by the Oracle Thin driver to work with Oracle RAC. As described in Oracle's Technical Note 235118.1, the Oracle Thin driver cannot guarantee that a transaction is initiated and concluded on the same Oracle RAC instance when the driver is configured for load balancing. As Oracle RAC requires that all database operations inside a global transaction be routed to the same Oracle instance, this known limitation means that you cannot use driver-level load balancing when using XA with Oracle RAC and therefore you cannot use a primary/primary RAC configuration.

Oracle RAC Failover with WebLogic Server

Although Oracle RAC offers JDBC connect-time failover features, for most configurations, BEA recommends using WebLogic JDBC MultiPools to handle failover instead.

Note: Transparent Application Failover (TAF) requires the use of the Oracle OCI driver. Because BEA requires the use of the Oracle Thin driver, TAF is not supported.

Environment

When using WebLogic Server with Oracle RAC, consider the following requirements:

- Hardware Requirements
- Software Requirements

Note: See the *WebLogic Platform Supported Configurations* documentation at `http://e-docs.bea.com/platform/suppconfigs/index.html` for the latest WebLogic Server hardware platform and operating system support, and for the Oracle RAC versions supported by WebLogic Server versions and service packs. See the Oracle documentation for hardware and software requirements for running the Oracle RAC software.

Hardware Requirements

A typical WebLogic Server/Oracle RAC system includes a WebLogic Server cluster, an Oracle RAC cluster, and hardware for shared storage.

WebLogic Server Cluster

The WebLogic Server cluster can be configured in many ways and with various hardware options. See *Using WebLogic Server Clusters* for more details about configuring a WebLogic Server cluster.

Oracle RAC Cluster

For the latest hardware requirements for Oracle RAC, see the Oracle RAC documentation. However, to use Oracle RAC with WebLogic Server, you must run Oracle RAC instances on robust, production-quality hardware. The Oracle RAC configuration must deliver database processing performance appropriate for reasonably-anticipated application load requirements. Unusual database response delays can lead to unexpected behavior during database failover scenarios.

Shared Storage

In an Oracle RAC configuration, all data files, control files, and parameter files are shared for use by all RAC instances. An high availability (HA) storage solution that uses one of the following architectures is recommended:

- Direct Attached Storage (DAS), such as a dual ported disk array or a Storage Area Network (SAN)
- Network Attached Storage (NAS)

For a complete list of supported storage solutions, see your Oracle documentation.

Software Requirements

To use WebLogic Server with Oracle RAC, you must install the following software on each RAC node:

- Operating system patches required to support Oracle RAC. See the release notes from Oracle for details.
- Oracle 9i or Oracle 10g database management system
- Clustering software for your operating system. See the Oracle documentation for supported clustering software and cluster configurations.
- Shared storage software, such as Veritas Cluster File System. Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

Note: See the *WebLogic Platform Supported Configurations* documentation at `http://e-docs.bea.com/platform/suppconfigs/index.html` for the latest WebLogic Server hardware platform and operating system support, and for the Oracle RAC versions supported by WebLogic Server versions and service packs. See the Oracle documentation for hardware and software requirements required for running the Oracle RAC software.

Configuration Considerations for Oracle

Once you have installed and configured Oracle RAC, you must configure the listener process for each RAC instance as described in the sections that follow. For information about installing and configuring your operating system and the Oracle software for Oracle RAC nodes see the Oracle documentation.

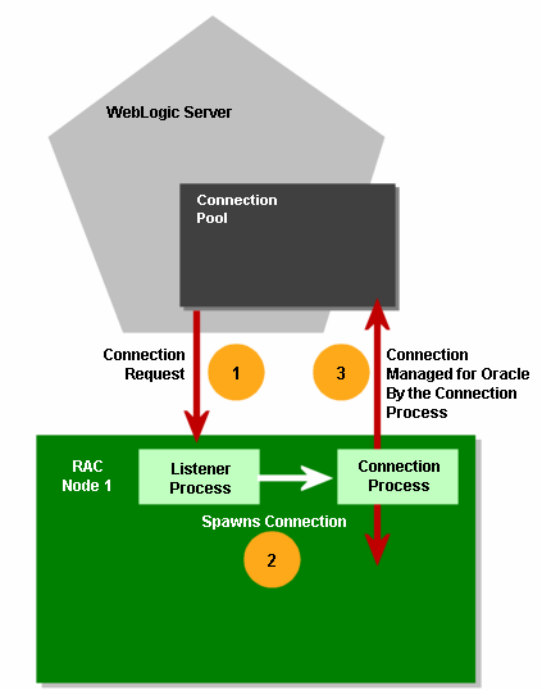
Configuring the Listener Process for Each Oracle RAC Instance

For Oracle RAC, the listener process establishes a communication pathway between a user process and an Oracle instance. When you use Oracle RAC with WebLogic Server, the user process is typically a JDBC connection pool.

When a connection pool is created, it attempts to create a pool of database connections for applications to borrow. If a pooled database connection becomes inoperative or if the connection pool is configured to grow in capacity, the connection pool attempts to create additional database connections up to the maximum specified in the configuration file. In all of these instances, the Oracle listener process handles the connection request on the Oracle RAC instance.

Figure A-2 shows the Oracle listener process functionality.

Figure A-2 Oracle Listener Process Functionality



To enable this functionality, you must configure the listener process for each RAC instance in the Oracle cluster. BEA requires that you configure a local listener on each RAC instance. Each database instance should be configured to register with its local listener only.

Oracle instances can be configured to register with the listener statically in the `listener.ora` file or registered dynamically using the instance initialization parameter `local_listener`, or both. BEA recommends using dynamic registration.

A listener can start either a shared dispatcher process or a dedicated process. When using with WebLogic Server, BEA recommends using dedicated processes.

Disabling Remote Listeners

Although Oracle RAC allows you to configure remote listeners to handle connection failover, remote listeners are typically too slow and BEA does not support their use. To disable remote listeners, delete any listed remote listeners in `spfile.ora` on each RAC node. For example:

```
*.remote_listener=''
```

BEA recommends using one of the following methods of handling failover instead.

- JDBC MultiPools (recommended)
- Oracle Thin driver connect-time failover (supported for some configurations)

Configuration Options in WebLogic Server with Oracle RAC

When using WebLogic Server with Oracle 9i RAC or Oracle 10g RAC, you must configure your WebLogic Domain so that it can interact with RAC instances and so that it performs as expected. The following sections describe configuration options and requirements:

- Choosing a WebLogic Server Configuration for Use with Oracle RAC
- Required JDBC Drivers
- Configuration Considerations for Failover
- Using MultiPools with Oracle RAC
- Using MultiPools with Global Transactions
- Using MultiPools without Global Transactions
- Using Connect-Time Failover with Oracle RAC

- Using Connect-Time Failover without Global Transactions
- Using Connect-Time Failover with Global Transactions

Choosing a WebLogic Server Configuration for Use with Oracle RAC

BEA supports several configuration options for using Oracle RAC with WebLogic Server:

- To connect to multiple Oracle 9i RAC or Oracle 10g RAC instances when using global transactions (XA), BEA recommends the use of transaction-aware WebLogic JDBC MultiPools, which support failover and load balancing, to connect to the RAC nodes. For more information see Using MultiPools with Global Transactions. You must either Weblogic Server 8.1 SP5 or later or have WebLogic Server 8.1 SP4 with the Oracle 10g RAC patch installed for this configuration. The WebLogic Server 8.1 SP4 Oracle 10g RAC patch is available at http://dev2dev.bea.com/wlserver/patch/wls81sp4_MP_OracleRAC_patch.html
- To connect to multiple Oracle 9i RAC or Oracle10g RAC instances when not using XA, BEA recommends the use of (non-transaction-aware) MultiPools to connect to the RAC nodes. Use the standard MultiPool configuration, which supports failover and load balancing. For more information see Using MultiPools without Global Transactions.
- To connect to multiple Oracle RAC nodes when MultiPools are not an option and when not using XA, use Oracle 9i RAC or Oracle 10g RAC with connect-time failover. Note that load balancing is not supported in this configuration. For more information see Using Connect-Time Failover without Global Transactions. You must have either Weblogic Server 8.1 SP5 or later or WebLogic Server 8.1 SP4 with the Oracle 9i RAC patch installed for this configuration. The WebLogic Server 8.1 SP4 Oracle 9i RAC patch is available at http://commerce.bea.com/d2d/wlplat81sp4_Oracle9iRAC_patch.jsp.
- To connect to multiple Oracle RAC nodes when MultiPools are not an option and when using XA, use Oracle 9i RAC with connect-time failover. Note that load balancing is not supported in this configuration due to known limitations of the Oracle Thin driver described in Oracle's Technical Note 235118.1. For more information see Using Connect-Time Failover with Global Transactions. You must have either Weblogic Server 8.1 SP5 or WebLogic Server 8.1 SP4 with the Oracle 9i RAC patch installed for this configuration. The WebLogic Server 8.1 SP4 Oracle 9i RAC patch is available at http://commerce.bea.com/d2d/wlplat81sp4_Oracle9iRAC_patch.jsp.

The following table may help you as you try to determine which configuration is right for your particular application:

Does Your Application Require				
Load Balancing?	**Failover?**	**Global Transactions (XA)?**	**JMS JDBC Store?**	
Y	Y	Y	N	See Using MultiPools with Global Transactions.
Y	Y	N	N	See Using MultiPools without Global Transactions.
N	Y	N	Y	See Using Connect-Time Failover without Global Transactions
N	Y	Y	N	See Using Connect-Time Failover with Global Transactions

Required JDBC Drivers

To use WebLogic Server with Oracle RAC, your WebLogic JDBC connection pools must use the Oracle JDBC Thin driver 10g to create database connections.

Configuration Considerations for Failover

Consider the following information when configuring for failover.

MultiPool-Managed Failover

MultiPools offer failover for global transactions without the limitations and known issues associated with a connection pool configuration with connect-time failover. For a description of MultiPool failover features, see Configuring and Using MultiPools.

With this configuration, pictured in Figure A-3, "MultiPool Configuration," on page A-13, you get:

- Faster failover controlled by the MultiPool
- Automatic failback by the WebLogic Server health monitor

The MultiPool handles failover for database connections when a RAC node becomes unavailable. When WebLogic Server tests a connection and the connection fails, it attempts to recreate the connection. If that attempt fails, the server disables the connection pool and routes connection requests to other connection pools (which correspond to other RAC nodes) in the MultiPool. WebLogic Server periodically tries to recreate the database connections in the disabled connection pool. When WebLogic Server is successful in recreating the connections, it next re-enables the connection pool and begins routing connection requests to the connection pool again. Because of the connection request routing and automatic health checking features, there is minimal delay in satisfying connection requests after a failure compared to when relying on the Oracle Thin driver connect-time failover configuration.

Connect-Time Failover

When MultiPools are not an option, WebLogic Server relies on the connect-time failover feature of the Oracle Thin driver to handle connection failover when a RAC instance becomes unavailable and a primary/primary configuration is not an option. When WebLogic Server tests a connection and the connection fails, the server replaces it by getting a new connection, and the driver again determines which RAC instance to use based on instance availability. The `CountOfTestFailuresTillFlush="1"` attribute helps to minimize the delay in delivering a connection to applications caused by the testing task. With this attribute setting, when a connection fails a connection test the first time, WebLogic Server automatically closes all connections in the connection pool. WebLogic Server replaces the connection with a new one, relying on the driver to determine to which node it should connect. In this case, the primary RAC node has failed, so the new connection is to the secondary RAC node. WebLogic Server tests the new connection before satisfying the request.

Delays During Failover

Occasionally, when one RAC node fails over to another, there may be a delay before the data associated with a transaction branch in progress on the now failed node is available throughout the cluster. This prevents incomplete transactions from being properly completed, which could further result in data locking in the database. To protect against the potential consequences of such a delay, WebLogic Server provides two configuration attributes that enable XA call retry for Oracle RAC: `XARetryDurationSeconds` and `XARetryIntervalSeconds`.

`XARetryDurationSeconds` controls the period of time during which WebLogic Server will repeatedly retry XA operations such as recover, commit and rollback for pending transactions. `XARetryIntervalSeconds` controls the frequency of the retry attempts within the established time period.

To enable XA call retries, add a value for `XARetryDurationSeconds` to all JDBC connection pools in your WebLogic domain that connect to an Oracle RAC instance. For example:

```
<JDBCConnectionPool
        Name="oracleRACPool"
        DriverName="oracle.jdbc.xa.client.OracleXADataSource"
        ...
        XARetryDurationSeconds=480
        />
```

Note: `XARetryDurationSeconds` is available in the Administration Console. To enable this feature, you must manually edit your config.xml file or change the configuration using the weblogic.Admin command line utility or a JMX program.

Use the following formula to determine the value for `XARetryDurationSeconds`:

`XARetryDurationSeconds` = (longest transaction timeout for transactions that use connections from the connection pool) + (delay before XIDs are available on all RAC nodes, typically less than 5 minutes)

For example, if your application sets the longest transaction timeout as 180 seconds, you should set `XARetryDurationSeconds` to 180 seconds + 300 seconds, for a total of 480 seconds.

Note: It is generally better to set `XARetryDurationSeconds` higher than minimally necessary to make sure that all transactions are completed properly. Setting the value higher than minimally required should not affect application performance during normal operations. The additional processing only affects transactions that have been prepared but have failed to complete.

You can also optionally set a value for `XARetryIntervalSeconds`. This value determines the time between XA retry calls. By default, the value is 60 seconds. Decreasing the value will decrease the amount of time between XA retry attempts. The default value should suffice in most cases.

Failure Handling Walkthrough for Global Transactions

What happens to inflight transactions to a database node if that node fails? When the primary Oracle RAC node fails? Does WebLogic Server support transparent failover? To answer these and other questions about how WebLogic Server handles failures, let's walk through the transaction processing steps and describe how a failure would be handled at each stage along the way.

The first stage at which a failure may occur is before the application calls for the transaction to be committed. If a database or RAC node fails at this stage, the application receives an exception

and must get a new connection and make a new attempt at processing the transaction. WebLogic Server does not support transparent failover.

If a failure occurs after the application has called for the transaction to be committed, the handling of any in-flight transaction depends upon whether the `PREPARE` operation is complete. If the `PREPARE` operation is not complete, the transaction manager rolls back the transaction and sends the application an exception for the failed transaction. If the `PREPARE` operation is complete, the transaction manager attempts to drive the in-flight transaction to completion using another node.

If a failure occurs during the `COMMIT` operation, the transaction manager attempts to retry the `COMMIT` operation several times. Note that the connection is blocked during these attempts. If the `COMMIT` operation is not successful during the first set of retry attempts, the application receives an exception. The transaction manager then continues to retry the `COMMIT` operation periodically until it is successful; if the transaction cannot be completed successfully within the abandon time period, the transaction is driven to completion heuristically.

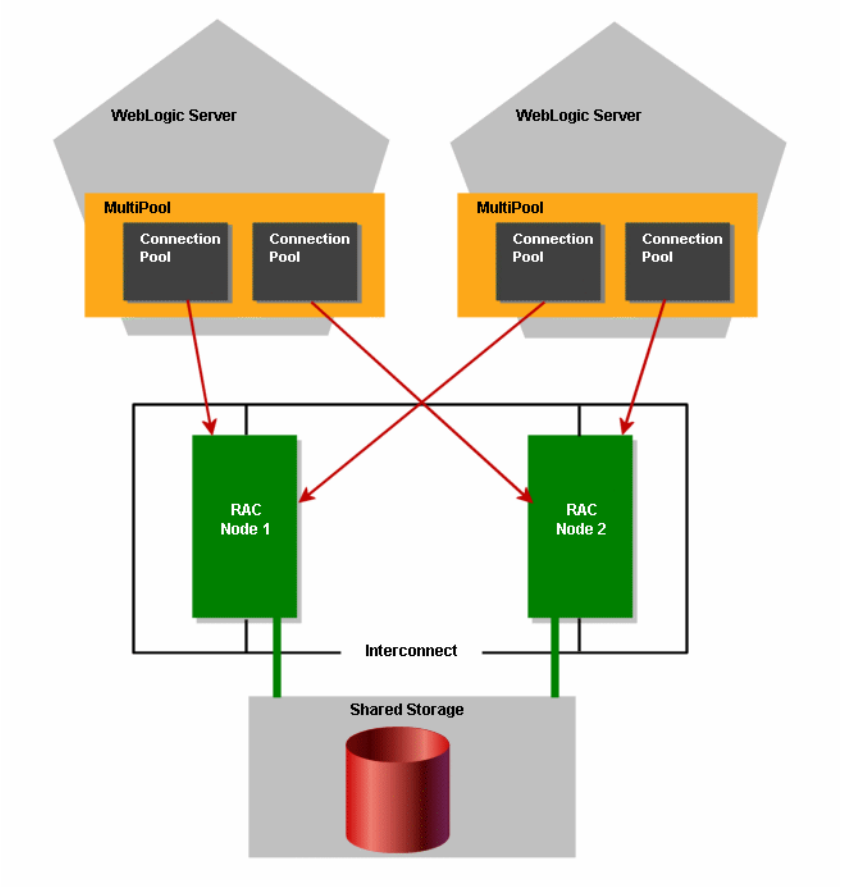
Using MultiPools with Oracle RAC

To connect WebLogic Server to multiple Oracle RAC nodes using MultiPools, first configure a JDBC connection pool for each RAC instance in your RAC cluster with the Oracle Thin driver. Then configure a MultiPool, using either the algorithm for load balancing or the algorithm for high availability, and add the connection pools to it.

Note: WebLogic Server does not support the use of MultiPools with JMS JDBC Stores. If your application makes use of JMS JDBC Stores, you must configure your JMS JDBC Store to use Oracle RAC with connect-time failover. For more information see Using Connect-Time Failover with Oracle RAC.

Figure A-3 shows a typical MultiPool configuration.

Figure A-3 MultiPool Configuration



You can use the Administration Console or any other means that you prefer to configure your domain, such as the `weblogic.Admin` command line utility, the WebLogic Scripting Tool (WLST), or a JMX program. For information about configuring a WebLogic JDBC MultiPool see "Configuring Multipools" in the *Administration Console Online Help*.

To use a database connection in this configuration, your applications look up a data source on the JNDI tree and then request a connection from the data source. The data source communicates with the MultiPool and the MultiPool determines which connection pool to use to satisfy the connection request based on the algorithm type specified in the configuration (that is, high availability or load balancing).

Attributes of a MultiPool

The MultiPool may have the following attributes, depending on the role of RAC in your system—load balancing or failover:

- `AlgorithmType="Load-Balancing"` or `AlgorithmType="High-Availability"`
  - With the Load-Balancing option, connection requests are distributed among available connection pools; with the High-Availability option, connection requests are served by the first available pool in the list. When a connection pool becomes defunct, connection requests are served by the next connection pool in the list.
- `FailoverRequestIfBusy="true"`
  - With the High-Availability algorithm, this attribute enables failover when all connections in a connection pool are in use.
- `HealthCheckFrequencySeconds="240"`
  - Controls how often WebLogic Server checks automatically disabled connection pools in a MultiPool to see if connections can be recreated and if the connection pool can be re-enabled. The default is 300 seconds. See "MultiPool Failover Enhancements" in *Programming WebLogic JDBC* for more details.

Using MultiPools with Global Transactions

In this configuration, a MultiPool "pins" a transaction to one and only one Oracle RAC instance and failover is handled at the MultiPool level when a RAC instance becomes unavailable. If there is a failure on a RAC instance before PREPARE, the operation is retried until the retry duration has expired. If there is a failure after PREPARE the transaction is failed over to another instance.

Rules for Connection Pools within a MultiPool Using Global Transactions

The following rules apply to the XA connection pools within a MultiPool:

- All the connection pools must be homogeneous. In other words, either all of them must be XA connection pools or none of them can be XA connection pools.
- If you choose to specify them, all XA-related attributes must be set to the same values for each connection pool. The attributes include the following:
  - `XARetryDurationSeconds`
  - `SupportsLocalTransaction`
  - `KeepXAConnTillTxComplete`

  - `NeedTxCtxOnClose`
  - `XAEndOnlyOnce`
  - `NewXAConnForCommit`
  - `RollbackLocalTxUponConnClose`
  - `RecoverOnlyOnce`
  - `KeepLogicalConnOpenOnRelease`

Notes:

- WebLogic Server 8.1 SP5 is certified to support Multipools with XA only on Oracle RAC. For information on supported versions of Oracle RAC, see Supported Database Configurations.
- If you are not using XA, BEA recommends the use of MultiPools for failover and load balancing across RAC instances, but the XA-specific configuration requirements above do not apply. For more information about configuring WebLogic JDBC MultiPools, see Configuring Multipools" in the *Administration Console Online Help*.

Required Attributes of Connection Pools within a MultiPool Using Global Transactions

Each connection pool within the MultiPool should have the following attributes:

- Oracle JDBC Thin driver 10g. For example:

```
DriverName="oracle.jdbc.xa.client.OracleXADataSource"
URL="jdbc:oracle:thin:@db_server1:1521:SNRAC1"
```

- `KeepXAConnTillTxComplete="true"`
  - Forces the connection pool to reserve a physical database connection and provide the same connection to an application throughout transaction processing until the distributed transaction is complete.
  - Required for proper transaction processing with Oracle RAC.
- `XARetryDurationSeconds="300"`
  - Enables the WebLogic Server transaction manager to retry XA recover, commit, and rollback calls for the specified amount of time.
- `CountOfTestFailuresTillFlush="1"`

  - Enables WebLogic Server to close all connections in the connection pool after the number of test failures that you specify to minimize the delay caused by further database testing. See "JDBC Connection Pool Testing Enhancements" in *Programming WebLogic JDBC* for more details about this attribute.
  - Minimizes the failover time when an Oracle RAC node fails.

- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the connection pool. See "Testing Connection Pools and Database Connections" in *Programming WebLogic JDBC* for more details about this attribute.
  - Required to enable failover to another RAC node.

- `TestTableName="name_of_small_table"` The name of the table used to test a physical database connection. For more details about this attribute, see "JDBC Connection Pool → Configuration →Connections" in the *Administration Console Online Help*.

Sample config.xml Code

An example of the connection pools, a WebLogic JDBC MultiPool, and an associated data source in the `config.xml` file would be:

```
<JDBCConnectionPool
        CapacityIncrement="1"
        ConnLeakProfilingEnabled="true"
        CountOfTestFailuresTillFlush="1"
        DriverName="oracle.jdbc.xa.client.OracleXADataSource"
        InitialCapacity="5"
        KeepXAConnTillTxComplete="true"
        MaxCapacity="100"
        Name="jdbcXAPool1"
        PasswordEncrypted="{3DES}lBifoTsg8fc="
        Properties="user=wlsqa"
        RefreshMinutes="1" SupportsLocalTransaction="true"
        Targets="WLSCluster"
        TestConnectionsOnReserve="true"
        TestTableName="dual"
        URL="jdbc:oracle:thin:@db_server1:1521:SNRAC1"
        XAEndOnlyOnce="true"
        XARetryDurationSeconds="300"
```

```
        XASetTransactionTimeout="true"/>
        XATransactionTimeout="302"/>
<JDBCConnectionPool
        CapacityIncrement="1"
        ConnLeakProfilingEnabled="true"
        CountOfTestFailuresTillFlush="1"
        DriverName="oracle.jdbc.xa.client.OracleXADataSource"
        InitialCapacity="5"
        KeepXAConnTillTxComplete="true"
        MaxCapacity="100"
        Name="jdbcXAPool2"
        PasswordEncrypted="{3DES}lBifoTsg8fc="
        Properties="user=wlsqa"
        RefreshMinutes="1"
        SupportsLocalTransaction="true"
        Targets="WLSCluster"
        TestConnectionsOnReserve="true"
        TestTableName="dual"
        URL="jdbc:oracle:thin:@db_server2:1521:SNRAC2"
        XAEndOnlyOnce="true"
        XARetryDurationSeconds="300"
        XASetTransactionTimeout="true"/>
        XATransactionTimeout="302"/>
<JDBCMultiPool
        Name="jdbcXAMultiPool1"
        PoolList="jdbcXAPool1,jdbcXAPool2"
        Targets="WLSCluster"
        AlgorithmType="Load-Balancing"/>
<JDBCTxDataSource
        JNDIName="jdbcXADataSource1"
        Name="jdbcXADataSource1"
        PoolName="jdbcXAMultiPool1"
        Targets="WLSCluster"/>
```

Note: Line breaks added for readability.

Using MultiPools without Global Transactions

The following sections describe a configuration that uses Oracle RAC with MultiPools in an application that does not require global transactions.

Attributes of Connection Pools within a MultiPool Not Using Global Transactions

Connection pools must have the following attributes:

- Oracle JDBC Thin driver 10g. For example:

  ```
  DriverName="oracle.jdbc.OracleDriver"
  URL="jdbc:oracle:thin:@db_server1:1521:SNRAC1"
  ```

- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the connection pool. See "Testing Connection Pools and Database Connections" in *Programming WebLogic JDBC* for more details about this attribute.
  - Required to enable failover and connection request routing within a MultiPool (effectively, failover to another RAC node).

- `TestTableName="name_of_small_table"`
  - The name of the table used to test a physical database connection. For more details about this attribute, see "JDBC Connection Pool →Configuration →Connections" in the *Administration Console Online Help*.

Sample config.xml Code

An example of the connection pools, a WebLogic JDBC MultiPool, and an associated data source in the `config.xml` file would be:

```
<JDBCConnectionPool
        Name="oracleRACPool_1"
        DriverName="oracle.jdbc.OracleDriver"
        InitialCapacity="5"
        MaxCapacity="100"
        Password="{3DES}I5fj3vh4+nI="
        Properties="user=SCOTT"
        RefreshMinutes="5"
        TestConnectionsOnReserve="true"
```

```
        TestTableName="dual"
        Targets="myWebLogicCluster"
        URL="jdbc:oracle:thin:@dbhost1:1521:dbservice"
        />
<JDBCConnectionPool
        Name="oracleRACPool_2"
        DriverName="oracle.jdbc.OracleDriver"
        InitialCapacity="5"
        LoginDelaySeconds="1"
        MaxCapacity="5"
        Password="{3DES}I5fj3vh4+nI="
        Properties="user=SCOTT"
        RefreshMinutes="5"
        TestConnectionsOnReserve="true"
        TestTableName="dual"
        PreparedStatementCacheSize="15"
        Targets="myWebLogicCluster"
        URL="jdbc:oracle:thin:@dbhost2:1521:dbservice"
        />
<JDBCMultiPool
        AlgorithmType="Load-Balancing"
        Name="MyJDBCMultiPool"
        HealthCheckFrequencySeconds="300"
        PoolList="oracleRACPool_1,oracleRACPool_2"
        Targets="myWebLogicCluster"
        />
<JDBCDataSource
        JNDIName="oracleRACDataSource"
        Name="oracleRACDataSource"
        PoolName="MyJDBCMultiPool"
        Targets="myWebLogicCluster"
        />
```

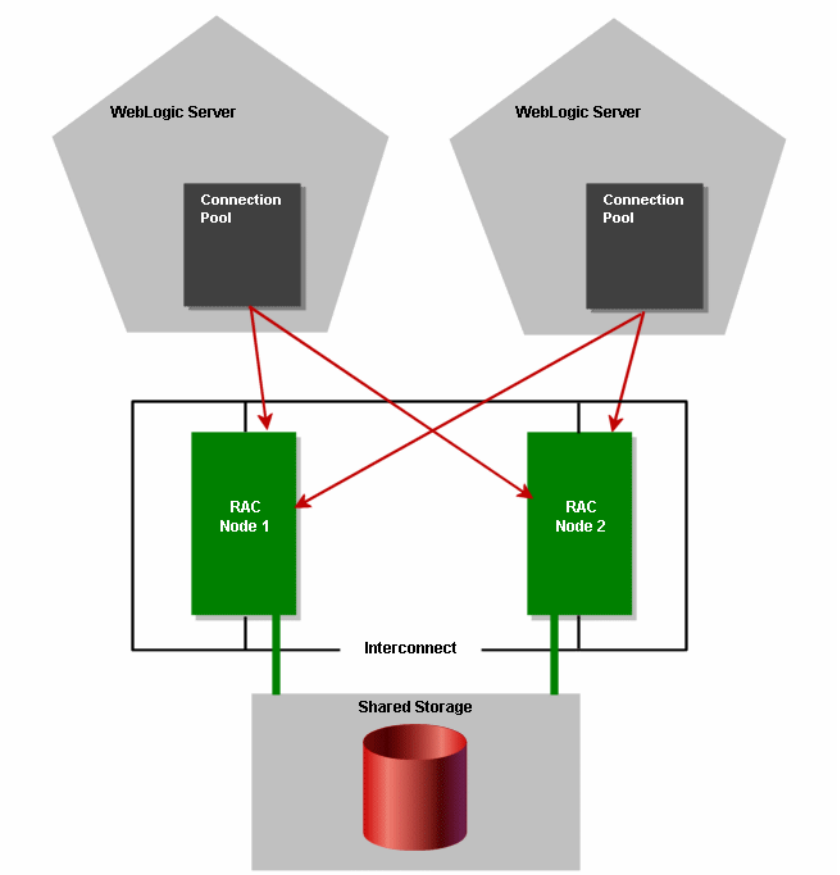
Note: Line breaks added for readability.

Using Connect-Time Failover with Oracle RAC

When MultiPools are not an option in your application (as when you are using a JMS JDBC Store, which does not support the use of MultiPools) and if load balancing is not required, you can configure your connection pools to use connect-time failover.

To connect WebLogic Server to multiple Oracle RAC nodes using connection pools configured for connect-time failover, configure a JDBC connection pool for each RAC instance in your RAC cluster with the Oracle Thin driver. Configure each connection pool to use connect-time failover, as described in the sections that follow. Figure A-4 shows an overview of the system.

Figure A-4 Connection Pool Configuration with Oracle Thin Driver Connect-Time Failover



You can use the Administration Console or any other means that you prefer to configure your domain, such as the `weblogic.Admin` command line utility, the Weblogic Scripting Tool (WLST), or a JMX program.

When connections are created in the connection pool, the Oracle Thin driver determines which Oracle RAC instance to use. When an application gets a connection, it looks up a data source on the JNDI tree and requests a connection from the data source. The underlying connection pool delivers one of the available connections from the pool.

The following sections describe configuration options and requirements:

- Using Connect-Time Failover without Global Transactions
- Using Connect-Time Failover with Global Transactions
- XA Considerations and Limitations with Oracle 9i RAC
- JMS Store Recovery with Oracle RAC

Using Connect-Time Failover without Global Transactions

The following sections describe a configuration which uses Oracle RAC's connect-time failover features to handle connection failures. With this configuration, in some failure cases, the failover time is as long as the TCP timeout, which can be several minutes, depending on your environment.

Attributes of a Connect-Time Failover Configuration without Global Transactions

To use this configuration, create JDBC connection pools in your WebLogic domain with the following attributes.

- Oracle JDBC Thin driver 10g configured for connect-time failover. For example:

```
DriverName="oracle.jdbc.OracleDriver"

URL="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=
(ADDRESS=(PROTOCOL=TCP)(HOST=dbhost1)(PORT=1521))
(ADDRESS=(PROTOCOL=TCP)(HOST=dbhost2)(PORT=1521))
(FAILOVER=on)(LOAD_BALANCE=off))(CONNECT_DATA=(SERVER=DEDICATED)
(SERVICE_NAME=dbservice)))"
```

- `CountOfTestFailuresTillFlush="1"`
  - Enables WebLogic Server to close all connections in the connection pool after the number of test failures that you specify to minimize the delay caused by further

database testing. See "JDBC Connection Pool Testing Enhancements" in *Programming WebLogic JDBC* for more details about this attribute.

  - Minimizes the failover time when an Oracle RAC node fails.

- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the connection pool. See "Testing Connection Pools and Database Connections" in *Programming WebLogic JDBC* for more details about this attribute.
  - Required to enable failover to another RAC node.
- `TestTableName="name_of_small_table"` The name of the table used to test a physical database connection. For more details about this attribute, see "JDBC Connection Pool \ Configuration \ Connections" in the *Administration Console Online Help*.

You can also optionally set a time-out value using the `ConnectionReserveTimeoutSeconds` attribute. This value determines the maximum time an application will wait for a connection to be made available. For example the following statement will set the time-out value to 120 seconds:

```
ConnectionReserveTimeoutSeconds="120"
```

Sample config.xml Code

```
<JDBCConnectionPool Name="jdbcPool2"
       Targets="JdbcRacCluster"
       DriverName="oracle.jdbc.OracleDriver"
       URL="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST = (ADDRESS =
       (PROTOCOL = TCP)(HOST=dbhost1)(PORT=1521))
       (ADDRESS=(PROTOCOL=TCP)(HOST=dbhost2)(PORT=1521))
       (FAILOVER=on) (LOAD_BALANCE=off))
       (CONNECT_DATA = (SERVER=DEDICATED) (SERVICE_NAME =dbservice)))"
       InitialCapacity="10"
       MaxCapacity="100"
       CapacityIncrement="1"
       Password="tiger"
       Properties="user=scott"
       PreparedStatementCacheSize="15"
       ConnLeakProfilingEnabled="true"
       TestTableName="dual"
```

```
        TestConnectionsOnReserve="true"
        CountOfTestFailuresTillFlush="1" />
<JDBCDataSource Name="jdbcDataSource2"
        Targets="JdbcRacCluster" `
        JNDIName="jdbcDataSource2"
        PoolName="jdbcPool2" />
```

Note: Line breaks added for readability.

Using Connect-Time Failover with Global Transactions

To use XA with a connect-time failover configuration, you must be using Oracle 9i RAC. When connections are created in the connection pool, the Oracle Thin driver determines which Oracle 9i RAC instance to use. When an application gets a connection, it looks up a data source on the JNDI tree and requests a connection from the data source. The underlying connection pool delivers one of the available connections from the pool.

With this configuration, in some failure cases, the failover time is as long as the TCP timeout, which can be several minutes, depending on your environment.

Attributes of a Connect-Time Failover Configuration with Global Transactions

To use this configuration, create JDBC connection pools in your WebLogic domain with the following attributes.

- Oracle JDBC Thin driver 10g configured for connect-time failover. For example:

  ```
  DriverName="oracle.jdbc.xa.client.OracleXADataSource"
  ```

  ```
  URL="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=
  (ADDRESS=(PROTOCOL=TCP)(HOST=dbhost1)(PORT=1521))
  (ADDRESS=(PROTOCOL=TCP)(HOST=dbhost2)(PORT=1521))
  (FAILOVER=on)(LOAD_BALANCE=off))(CONNECT_DATA=(SERVER=DEDICATED)
  (SERVICE_NAME=dbservice)))"
  ```

- `KeepXAConnTillTxComplete="true"`
  - Forces the connection pool to reserve a physical database connection and provide the same connection to an application throughout transaction processing until the distributed transaction is complete.
  - Required for proper transaction processing with Oracle RAC.
- `XARetryDurationSeconds="300"`

  - Enables the WebLogic Server transaction manager to retry XA recover, commit, and rollback calls for the specified amount of time.

    Note: This attribute is not yet available in the Administration Console and must be set using a method such as the `weblogic.Admin` command line utility, the Weblogic Scripting Tool (WLST), or a JMX program

  - Resolves issues described in Oracle's bugs 3428146 and 395790. In some failure conditions, there is a window of time in which transaction IDs are not available across the RAC cluster, which prevents incomplete transactions from being properly completed, which further results in data locking in the database.

- `CountOfTestFailuresTillFlush="1"`
  - Enables WebLogic Server to close all connections in the connection pool after the number of test failures that you specify to minimize the delay caused by further database testing. See "JDBC Connection Pool Testing Enhancements" in *Programming WebLogic JDBC* for more details about this attribute.
  - Minimizes the failover time when an Oracle RAC node fails.
- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the connection pool. See "Testing Connection Pools and Database Connections" in *Programming WebLogic JDBC* for more details about this attribute.
  - Required to enable failover to another RAC node.
- `TestTableName="name_of_small_table"` The name of the table used to test a physical database connection. For more details about this attribute, see "JDBC Connection Pool \ Configuration \ Connections" in the *Administration Console Online Help*.

You can also optionally set a time-out value using the `ConnectionReserveTimeoutSeconds` attribute. This value determines the maximum time an application will wait for a connection to be made available. For example the following statement will set the time-out value to 120 seconds:

```
ConnectionReserveTimeoutSeconds="120"
```

Sample config.xml Code

An example of the connection pool and associated data source in the `config.xml` file would be:

```
<JDBCConnectionPool
        Name="oracleRACPool"
```

```
        DriverName="oracle.jdbc.xa.client.OracleXADataSource"
        InitialCapacity="5"
        LoginDelaySeconds="1"
        MaxCapacity="5"
        Password="{3DES}I5fj3vh4+nI="
        Properties="user=SCOTT"
        CountOfTestFailuresTillFlush="1"
        KeepXAConnTillTxComplete="true"
        XARetryDurationSeconds="300"
        RefreshMinutes="5"
        TestConnectionsOnReserve="true"
        TestTableName="dual"
        XASetTransactionTimeout="true"
        StatementCacheSize="15"
        Targets="myWebLogicCluster"
        URL="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=
        (ADDRESS=(PROTOCOL=TCP)(HOST=dbhost1)(PORT=1521))
        (ADDRESS=(PROTOCOL=TCP)(HOST=dbhost2)(PORT=1521))
        (FAILOVER=on)(LOAD_BALANCE=off)(CONNECT_DATA=(SERVER=DEDICATED)
        (SERVICE_NAME=dbservice)))"
        />
<JDBCTxDataSource
        JNDIName="oracleRACDataSource"
        Name="oracleRACDataSource"
        PoolName="oracleRACPool"
        Targets="myWebLogicCluster"
        />
```

Note: Line breaks added for readability.

XA Considerations and Limitations with Oracle 9i RAC

When using XA (global transactions) with Oracle 9i RAC, consider the following requirements and limitations:

- Required JDBC Driver Configuration for Use with XA
- Oracle 9i RAC XA Requirements

- Known Limitations When Using Oracle RAC with WebLogic Server
- Known Issue Occurring After Database Server Crash

Required JDBC Driver Configuration for Use with XA

In this configuration, you must use the Oracle Thin driver connect-time failover to create database connections as described in Using Connect-Time Failover without Global Transactions.

Oracle 9i RAC XA Requirements

Oracle 9i RAC has the following requirements when using XA.

A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the RAC Cluster

Global transactions must be initiated, prepared, and concluded in the same instance of the RAC cluster. WebLogic Server connection pools manage this for you when you set `KeepXAConnTillTxComplete="true"` in the JDBC connection pool configuration.

Note: WebLogic Server relies on the connect-time failover feature in the Oracle Thin driver to work with Oracle RAC. As described in Oracle's Technical Note 235118.1, the Oracle Thin driver cannot guarantee that a transaction is initiated and concluded on the same RAC instance when the driver is configured for load balancing. As Oracle RAC requires that all database operations inside a global transaction be routed to the same Oracle instance, this known limitation means that you cannot use connect-time load balancing when using XA with Oracle RAC and, therefore, you cannot use a primary/primary RAC configuration.

Transaction IDs Must Be Unique Within the RAC Cluster

When using global transactions, transaction IDs (XIDs) must be unique within the RAC cluster. However, neither the Oracle Thin driver nor an Oracle RAC instance can determine if an XID is unique within the RAC cluster. Transactions with the same XID can execute SQL code on different instances of the RAC cluster without any exception.

The WebLogic Server Transaction Manager generates unique transaction IDs. However, in some failover scenarios, a transaction can continue on a RAC instance other than the originating instance, which can cause data inconsistencies. See Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions.

Known Limitations When Using Oracle RAC with WebLogic Server

The following sections describe known issues and limitations when using XA and WebLogic Server with Oracle RAC:

- Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions
- Potential for Data Deadlocks in Some Failure Scenarios
- Potential for Transactions Completed Out of Sequence

Note: Some of these limitations are also described in Oracle's bug numbers 3428146 and 395790. Contact Oracle for more information about these issues.

Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions

In some failure conditions, when MultiPools are not being used, transaction processing (data changes) that occurred on a RAC instance other than the instance on which a transaction was initiated ***will be lost without any notification or exception***.

For example, consider the following WebLogic Server configuration:

- A WebLogic cluster containing two servers: `server1` and `server2`.
- A JDBC data source `ds1` targeted to the cluster. (Identical instances of the data source are present on all instances in the cluster.)
- A JDBC connection pool `cp1` is configured to connect to an Oracle RAC cluster with connect-time failover enabled. `RAC1` is set as the primary RAC instance; `RAC2` is set as the secondary RAC instance. `Cp1` is targeted to the cluster. (Identical instances of the connection pool are present on all nodes in the WebLogic cluster.)

In the following scenario, some data changes will be lost:

1. Network connectivity between `server2` and `RAC1` is lost, which causes database connections in `cp1` on `server2` to fail over to `RAC2`. The same connection pool on `server1` still has connections to `RAC1`.
2. On `server1`, an application starts a transaction and uses a database connection from `cp1` (a connection to `RAC1`) to make data changes.
3. The application invokes an EJB on `server2`, which uses a database connection from `cp1` on `server2` (a connection to `RAC2`) to make data changes.

4. The application completes the transaction on `server1`.

Result: Data changes on RAC1 are committed. ***Data changes on RAC 2 are ignored***. The WebLogic Server transaction manager calls prepare and commit on the resource. In this case, because the connection pools have the same name, they are considered to be the same resource, so the calls are made on only one instance of the connection pool. Because the connection pools contain connections to different RAC instances, the data changes are committed on one RAC instance, but the changes on the other RAC instance are lost.

Workaround: Provide redundant network hardware between the WebLogic Server instance and the Oracle RAC instance to avoid the network failure.

Potential for Data Deadlocks in Some Failure Scenarios

There is a window of time in which transaction IDs are not available across the RAC cluster. Because of this known Oracle limitation, after some failure conditions, some incomplete transactions cannot be properly completed, which can result in deadlocks in the database. BEA has provided a patch to work around this known issue in 8.1 SP4, and recommends that customers using WebLogic Server and XA with Oracle RAC migrate to WebLogic Server 8.1 SP4 and use this patch.

- For more details, see "Patch to Support Use with Oracle 9i RAC" in the *WebLogic Server8.1 Release Notes*.

Potential for Transactions Completed Out of Sequence

When using the Oracle DataBase Control, the order of transaction processing is not guaranteed. For example, if you implement a web service that uses DataBase Control do the following transaction sequence:

1. Create a table
2. Insert record 1
3. Insert record 2
4. Insert record 3
5. Select records

If the primary node goes down momentarily after the table is created, it is possible that transactions submitted to the database are performed out of sequence.

Known Issue Occurring After Database Server Crash

If, while a transaction is being processed, the database server instance crashes after the `PREPARE` operation is complete but before the results of that operation have been written to the transaction log, a `COMMIT` call from a client for that transaction may hang for several minutes and possibly until the TCP timeout period has expired. The window of time in which this might occur is small and the problem occurs rarely. There is no workaround for the issue at this time.

JMS Store Recovery with Oracle RAC

If you are using a JMS JDBC Store with Oracle RAC, there are features and limitations to consider that concern Oracle RAC node failover. See the following sections:

- Configuring a JMS JDBC Store for Use with Oracle RAC
- Automatic Retry
- Manual Retry
- Alternative: JMS File Store

Configuring a JMS JDBC Store for Use with Oracle RAC

The way that a JMS JDBC Store works limits the options you have for configuring one for use with Oracle RAC. A JMS JDBC Store holds on to a connection until that connection fails, at which point it grabs the next connection and repeats the process. Therefore you cannot implement load balancing with a JMS JDBC Store. Also, JMS JDBC Stores do not support MultiPools. The only remaining configuration option is to use connect-time failover without global transactions. For more information about this configuration option, see Using Connect-Time Failover without Global Transactions.

Automatic Retry

JMS has a limited connection retry mechanism which enables it to silently react to the failure of the RAC node that hosts its database connection. If the database has experienced either a minor network 'hiccup' or a RAC database has failed over to another node, the second connection attempt (the retry) will succeed to the next RAC node.

The time within which this retry is attempted and the number of retries attempted are limited to minimize the negative effects that an extended connection retry time could cause. If the database connection remains unavailable for a long period of time, the delay can impede the ability of JMS

to properly continue its processing (for example, to maintain proper message ordering). Also, the transaction manager could declare the JMS resource of a transaction to be dead if there is not enough processing progress made within this time period, or out-of memory conditions could arise. There are system-level tuning guidelines that can help minimize the RAC failover time frame which is critical to the success of the automatic retry.

The tight loop on the automatic retry is particularly important when JMS processing occurs with transactions. If an I/O failure occurs in the JMS JDBC Store, the store record is in an unknown state which will put the message itself in an unknown state. To prevent the message from being committed in this unknown state, JMS will mark the transaction associated with the message as a "failedTransaction". Any future attempts by the transaction manager to finishing committing the message will cause JMS to throw a `javax.transaction.xa.XAException` with an errorCode set to `XAException.XAER_RMERR`. This exception is an indication to the transaction manager that a transient error has occurred in the resource manager (JMS) and that the transaction manager should retry commit processing. The retry logic provides a second attempt to establish the connection before JMS communicates any failure to the upper layer which would translate into an RMERR. If the RMERR is generated, then the only way to recover the message and complete the transaction is to restart the JMS Server manually. See "Manual Retry".

Note: In certain scenarios, it is not possible to restart JMS server during runtime so, you need to restart WebLogic Server. If restarting WebLogic Server is not acceptable, you can choose File Store instead of JDBCStore. For more information, see "JMS Store Tasks" in JMS: Configuring in *Administration Console Online Help*.

The automatic retry logic is currently governed by an option on WebLogic Server as follows:

```
-Dweblogic.jms.store.JMSJDBCIORetryDelay=X
```

Where `X` is the number of seconds that should elapse before the connection to the database is retried. The default is one second. This value is restricted to the range `0` to `15`, and the retry will only be attempted once. If a failure occurs on the second attempt, an exception will be propagated up the call stack and a manual restart will be required to recover the messages associated with the failed transaction.

Note: If you continue to encounter `XAException.XAER_RMERR`, try setting `JMSJDBCIORetryDelay` to a higher number.

Manual Retry

In certain test scenarios, empirical data has shown that it can take an extended period of time for everything to be completely transferred and operational (including transactional data) on a new RAC node once a particular node fails. If this exceeds the maximum elapsed time for the JMS

automatic retry logic, then the current design expects that some type of manual intervention will occur to restart the JMS server so that it will begin processing again and deliver/recover messages in accordance with its specification and configuration. Below are two methods in which a JMS server can be manually restarted.

- Restart the JMS Server via the Administration Console:
  a. Click the affected JMS server in the left pane.
  b. In the right pane, click the Targets tab.
  c. Select None for the target of this server and click Apply.
  d. Re-select the original targets and click Apply.
- Restart the JMS Server via JMX:
  a. Get the JMS server MBean name using

```
java weblogic.Admin -url t3://localhost:7001 -username weblogic
-password weblogic get -type JMSServer -property Name
```

  b. Similarly, get the Server mbean name.
  c. To removeDeployment or addDeployment, use the following commands:

```
java weblogic.Admin -url t3://localhost:7001 -username weblogic
-password weblogic invoke -mbean "Server mbean name from above
command" -method removeDeployment "jms server mbean name from above
command"
```

```
java weblogic.Admin -url t3://localhost:7001 -username weblogic
-password weblogic invoke -mbean "Server mbean name from above
command" -method addDeployment "jms server mbean name from above
command"
```

  Successful execution returns the following message:

```
{MBeanName="your_domain_name:Name=your_target_server,Type=Server"{ad
dDeployment=true}}  Ok
```

  The application must then wait for the JMS server to complete its restart prior to any client resuming its message processing requests.

Alternative: JMS File Store

Generic file system anomalies can affect the availability of the file store (e.g., a disk crash, disk full, etc.). To mitigate these issues, use a data redundancy scheme such as dual-ported SCSI disks, a RAID array, or perhaps a SAN.

For more information, see the following documents:

- "JMS Store Tasks" in the *JMS: Configuring* section of the Administration Console Online Help
- "File Store Tuning" in *BEA WebLogic JMS Performance Guide*