BEA WebLogic Server™

and BEA WebLogic Express™

Programming WebLogic JDBC

BEA WebLogic Server Version 6.1
Document Date: August 5, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JDBC

Part Number	Document Date	Software Version
N/A	August 5, 2004	BEA WebLogic Server 6.1

Contents

About This Document

1. Introduction to WebLogic JDBC

5. Using WebLogic Multitier JDBC Drivers

6. Using Third-Party Drivers with WebLogic Server

7. Using dbKona

8. Testing JDBC Connections and Troubleshooting

About This Document

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

The document is organized as follows:

- Chapter 1, “Introduction to WebLogic JDBC,” introduces the JDBC components and JDBC API.
- Chapter 2, “Administration and Configuration for WebLogic JDBC,” describes how to administer JDBC in the WebLogic Server and describes the Administration Console.
- Chapter 3, “Performance Tuning Your JDBC Application,” describes how to obtain the best performance from JDBC applications.
- Chapter 4, “Configuring WebLogic JDBC Features,” describes how to use JDBC components with WebLogic Server Java applications.
- Chapter 5, “Using WebLogic Multitier JDBC Drivers,” describes how to set up your WebLogic RMI driver and JDBC clients to use with WebLogic Server.
- Chapter 6, “Using Third-Party Drivers with WebLogic Server,” describes how to set up and use third-party drivers with WebLogic Server.
- Chapter 7, “Using dbKona,” describes the dbKona classes that provide high-level database connectivity to Java applications.
- Chapter 8, “Testing JDBC Connections and Troubleshooting,” describes troubleshooting tips when using JDBC with WebLogic Server.

Audience

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server.

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;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`

Convention	Usage

|	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 introduce the JDBC components and JDBC API:

Overview of JDBC

JDBC is a Java API for executing SQL statements. The API consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.

JDBC is a *low-level* interface, which means that it is used to invoke (or call) SQL commands directly. In addition, JDBC is a base upon which to build higher-level interfaces and tools, such as JMS and EJB.

Overview of JDBC Drivers

JDBC drivers implement the interfaces and classes of the JDBC API. BEA provides a variety of options for database access using the JDBC API specification. These options include two-tier JDBC drivers, including WebLogic jDrivers for the Oracle, Microsoft SQL Server, and Informix database management systems (DBMS), and multitier drivers that work with WebLogic Server as an intermediary between a client application and the DBMS.

Types of JDBC Drivers

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

- *Two-tier drivers* that provide database access directly between a java application and the database. WebLogic Server uses a DBMS vendor-specific JDBC driver to connect to a back-end database, such as the WebLogic jDrivers for Oracle, Informix and Microsoft SQL Server.
- *Multitier drivers* that provide vendor-neutral database access. A Java client application can use a multitier driver to access any database configured in WebLogic server. BEA offers three multitier drivers—RMI, Pool, and JTS.

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

Table of Drivers

The following table summarizes the drivers that WebLogic Server uses.

Table 1-1 JDBC Drivers

Driver Tier	Type and Name of Driver	Database Connectivity	Documentation Sources
Two-tier (non-XA)	Type 2 (native .dll): ■ WebLogic jDriver for Oracle ■ Third-party drivers Type 4 (all Java) ■ WebLogic jDrivers for Informix and Microsoft SQL Server ■ Third-party drivers, including: Oracle Thin Sybase jConnect DB2 Informix JDBC	Between WebLogic Server and DBMS.	Programming WebLogic JDBC (this document) *Administration Guide*, "Managing JDBC Connectivity" Installing and Using WebLogic jDriver for Oracle Installing and Using WebLogic jDriver for Informix Installing and Using WebLogic jDriver for Microsoft SQL Server
Two-tier (XA)	Type 2 (native `.dll`) ■ WebLogic jDriver for Oracle XA	Between WebLogic Server and DBMS in distributed transactions.	Programming WebLogic JTA *Administration Guide*, "Managing JDBC Connectivity"
Multitier	Type 3 ■ RMI Driver ■ Pool Driver ■ JTS	Between client and WebLogic Server. The RMI driver replaces the deprecated t3 driver. The JTS driver is used in distributed transactions. The Pool and JTS drivers are server-side only.	Programming WebLogic JDBC (this document)

Description of JDBC Drivers

The following sections describe in detail the JDBC drivers introduced in Table 1-1 JDBC Drivers.

WebLogic Server JDBC Two-Tier Drivers

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

WebLogic jDriver for Oracle

BEA's Type 2 JDBC driver for Oracle, 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 Installing and Using WebLogic jDriver for Oracle at `http://e-docs.bea.com/wls/docs61oracle/index.html`.

WebLogic jDriver for Microsoft SQL Server

BEA's WebLogic jDriver for Microsoft SQL Server, included in the WebLogic Server 6.1 distribution, is a pure-Java, Type 4 JDBC driver that provides connectivity to Microsoft SQL Server. For more information, see Installing and Using WebLogic jDriver for Microsoft SQL Server at `http://e-docs.bea.com/wls/docs61/mssqlserver4/index.html`.

WebLogic jDriver for Informix

BEA's WebLogic jDriver for Informix, included in the WebLogic Server 6.1 distribution, is a pure-Java, Type 4 JDBC driver that provides connectivity to the Informix DBMS. For more information, see Installing and Using WebLogic jDriver for Informix at `http://e-docs.bea.com/wls/docs61/informix4/index.html`.

WebLogic Server JDBC Multitier Drivers

The following sections describe the WebLogic multitier JDBC drivers that provide database access to the client. For more information on these drivers, see Using WebLogic Multitier Drivers in *Programming WebLogic JDBC* at `http://e-docs.bea.com/wls/docs61/jdbc/rmidriver.html`.

WebLogic Pool Driver

The WebLogic Pool driver enables utilization of connection pools from server-side applications such as HTTP servlets or EJBs.

WebLogic RMI Driver

The WebLogic RMI driver is a multitier, Type 3, Java Data Base Connectivity (JDBC) driver that runs in WebLogic Server and can be used with any two-tier JDBC driver to provide database access. Additionally, when configured in a cluster of WebLogic Servers, the WebLogic RMI driver can be used for clustered JDBC, allowing JDBC clients the benefits of load balancing and failover provided by WebLogic Clusters.

WebLogic JTS Driver

The WebLogic JTS driver is a multitier, Type 3, JDBC driver used in distributed transactions across multiple servers with one database instance. The JTS driver is more efficient than the WebLogic jDriver for Oracle XA driver when working with only one database instance because it avoids two-phase commit.

Third-Party Drivers

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

- Are thread-safe
- Are EJB accessible; can implement transaction calls in JDBC

In addition, WebLogic Server multitier drivers only support JDBC API and third-party drivers that provide functionality beyond non-standard JDBC calls.

Cloudscape

An evaluation copy of this pure-Java DBMS from Cloudscape is included with your WebLogic Server distribution. A JDBC driver to access the Cloudscape DBMS is also included. This DBMS is used extensively in the code examples that are also included in the distribution. You may use this DBMS for testing and development if you do not have another DBMS available. There are limitations on the quantity of data that may be stored using this evaluation version.

For additional information, see Using the Cloudscape Database with WebLogic.

Sybase jConnect Driver

The two-tier Sybase jConnect Type 4 driver is bundled with your WebLogic Server distribution. This driver is provided for your use without charge. For information on using this driver with WebLogic Server, see “Using Third-Party Drivers with WebLogic Server” on page 6-1.

Oracle Thin Driver

The two-tier *Oracle Thin* Type 4 driver provides connectivity from WebLogic Server to Oracle DBMS. For information on using this driver with WebLogic Server, see “Using Third-Party Drivers with WebLogic Server” on page 6-1.

Overview of Connection Pools

Multitier drivers use WebLogic Server to access *connection pools* that provide ready-to-use pools of connections to your DBMS. Since these database connections are already established when the connection pool starts up, the overhead of establishing database connections is eliminated. You can utilize connection pools from server-side applications such as HTTP servlets or EJBs using the WebLogic Pool

driver or from stand-alone Java client applications using the WebLogic RMI driver. This section provides an introduction to connection pools. For more detailed information, see "Using Connection Pools" on page 4-1.

Connection pools require a two-tier JDBC driver to make the connection from WebLogic Server to the DBMS. This two-tier driver can be one of the WebLogic jDrivers or a third-party JDBC driver, such as the Sybase jConnect driver, which is bundled with the WebLogic distribution. The following table summarizes the advantages to using connection pools.

Table 1-2 Advantages to Using Connection Pools

Connection Pools Provide These Advantages. . .	With This Functionality . . .
Save time, low overhead	Making a DBMS connection is very slow. With connection pools, connections are established and available to users. The alternative is for application code to make its 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 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 like EJBs, pools are mandatory so WebLogic Server controls the JDBC connection. This ensures your EJB transactions are committed or rolled back correctly and completely.

Using Connection Pools with Server-side Applications

For database access from server-side applications, such as HTTP servlets, use the WebLogic Pool driver. For two-phase commit transactions, use the WebLogic Server JDBC/XA driver, WebLogic jDriver for Oracle/XA. For transactions distributed across multiple servers with one database instance, use the JTS driver. You can also access connection pools using the Java Naming and Directory Interface (JNDI) and a DataSource object.

Using Connection Pools with Client-side Applications

BEA offers the RMI driver for client-side, multitier JDBC. The RMI driver has the advantage of providing a standards-based approach using the Java 2 Enterprise Edition (J2EE) specifications. For new deployments, BEA recommends that you use the RMI driver, because the t3 client services are deprecated in this release.

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

Overview of MultiPools

JDBC MultiPools provide the option of choosing either the high availability or load balancing algorithm to enhance database connectivity. MultiPools are a “pool of pools” that allow a configurable algorithm for choosing among its list of pools, the pool that will be selected to provide the connection. For more information, see “Using MultiPools” on page 4-16.

Choosing the MultiPool Algorithm

You have the option of setting up a MultiPool in either of these ways:

- 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.

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 more information about JDBC objects in a cluster, see "JDBC Connections" in *Using WebLogic Server Clusters* at `http://e-docs.bea.com/wls/docs61/cluster/overview.html#jdbc`.

Overview of DataSources

DataSource objects provide a way for JDBC clients to obtain a DBMS connection. A DataSource is an interface between the client program and the connection pool. Each data source requires a separate DataSource object, which may be implemented as a DataSource class that supports either connection pooling or distributed transactions. For more information, see "Configuring and Using DataSources" on page 4-20.

JDBC API

To create a JDBC application, use the *java.sql* API. The API allows you to create the class objects necessary to establish a connection with a data source, send queries and update statements to the data source, and process the results.

WebLogic JDBC Interface Definitions

The following table lists JDBC interfaces frequently used with WebLogic Server. For a complete description of all JDBC interfaces, see the `java.sql` or `weblogic.jdbc` Javadoc.

JDBC Interface	Description
Driver	Sets up a connection between a driver and a database, and also gives information about the driver or information about making a connection to the database.The interface that every driver class must implement.
DataSource	Represents a particular DBMS or other data source. Used to establish a connection with a data source.
Statement	Sends simple SQL statements, with no parameters, to a database.
PreparedStatement	Inherits from Statement. Used to execute a pre-compiled SQL statement with or without IN parameters.
CallableStatement	Inherits from PreparedStatement. Used to execute a call to a database stored procedure; adds methods for dealing with OUT parameters.
ResultSet	Contains the results of executing an SQL query. It contains the rows that satisfy the conditions of the query.
ResultSetMetaData	Provides information about the types and properties of the columns in a ResultSet object.

JDBC Interface	Description
DataBaseMetaData	Provides information about a database as a whole. Returns either single values or a result set.
Clob	A built-in type that stores a Character Large Object as a column value in a row of a database table.
Blob	A built-in type that stores a Binary Large Object as a column value in a row of a database table.

For information about these interfaces when using WebLogic jDriver for Oracle, see Installing and Using WebLogic jDriver for Oracle at `http://e-docs.bea.com/wls/docs61/oracle/index.html`.

JDBC 2.0

WebLogic Server uses JDK 1.3, which supports JDBC 2.0.

Limitations

Please be aware of the following limitations:

- You cannot use Batch updates (`addBatch()`) with the `callableStatement or preparedStatement` SQL statements when using the RMI driver in conjunction with the WebLogic jDriver for Oracle or third-party 2-Tier drivers.

Platforms

Supported platforms vary by vendor-specific DBMSs and drivers. For current information, see BEA WebLogic Server Platform Support at `http://e-docs.bea.com/wls/certifications/certs_610/index.html`.

2 Administration and Configuration for WebLogic JDBC

This section provides an overview of the JDBC administrative tasks related to BEA WebLogic Server.

For additional information, see

- Managing JDBC Connectivity in the *Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html. Describes how to use the Administration Console and command-line interface to configure and manage connectivity.
- Administration Console Online Help at http://e-docs.bea.com/wls/docs61/ConsoleHelp/index.html. Describes how to use the Administration Console to set specific configuration tasks.
- "Configuring WebLogic JDBC Features" on page 4-1. Describes how to use the JDBC API to configure connectivity.

Configuring JDBC

The WebLogic Console provides the interface you use to enable, configure, and monitor features of the WebLogic Server, including JDBC. To invoke the Administration Console, refer to the procedures described in Configuring WebLogic Servers and Clusters in the Administration Guide at http://e-docs.bea.com/wls/docs61/adminguide/config.html. The attributes define the JDBC environment that includes:

- Connection Pools
- MultiPools
- DataSources

Configuring Connection Pools

You use the Administration Console to configure a connection pool, which includes defining the attributes and connection parameters, cloning pools, and assigning connection pools to a server or domain. For more information regarding connection pools, see “Using Connection Pools” on page 4-1 and for configuring for database connectivity, see Managing JDBC Connectivity in the Administration Guide.

Configuring MultiPools

You define, or name, a MultiPool in the Administration Console and then determine which of the previously defined connection pools will constitute a specific MultiPool. All of the connections in a particular *connection pool* are identical, that is, they are attached to a single database with the same user, password and connection properties. With *MultiPools*, however, the connection pools within a MultiPool may be associated with different DBMSs. You have the option of setting the search methodology by selecting either the load balancing or the high availability algorithm behavior.

For more information regarding uses for MultiPools, see “Using MultiPools” on page 4-16 and for configuring for database connectivity, see Managing JDBC Connectivity in the Administration Guide.

Configuring DataSources and TxDataSources

As with Connection Pools and MultiPools, you create the DataSource objects in the Administration Console. Data Source objects can be defined with (TxDataSource) or without (DataSource) transaction services. You configure Connection Pools and MultiPools before you define the DataSource pool name attribute. For DataSource objects in local and distributed transactions, see Managing JDBC Connectivity in the *Administration Guide* and "Configuring WebLogic JDBC Features" on page 4-1.

Monitoring JDBC Connectivity

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

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/docs61/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 the Monitoring a WebLogic Domain and Managing JDBC Connectivity sections of the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/index.html`.

For information about using MBeans, see Programming WebLogic JMX Services at `http://e-docs.bea.com/wls/docs61/jmx/index.html`.

CHAPTER

3 Performance Tuning Your JDBC Application

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

Overview of JDBC Performance

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

WebLogic Performance-Enhancing Features

WebLogic has several features that enhance performance for JDBC applications.

How Connection Pools Enhance Performance

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

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

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

Caching Data

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

- Read-Only Entity Beans at http://e-docs.bea.com/wls/docs61/ejb/EJB_environment.html
- JNDI in a Clustered Environment at http://e-docs.bea.com/wls/docs61/jndi/jndi.html

Designing Your Application for Best Performance

The large majority of the performance to be gained or lost in a DBMS application is not in the application language, but in 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.

Below 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.

You should 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 hundred employees represented in the tables than to alter each table 100 times, once for each employee.

Understanding set-based methodology can be very useful. 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. This is a valid question, but there is no easy answer. 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. The actual count may change when the query is issued if there has been any other DBMS activity which alters the relevant data.

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

Your application should tailor its queries to be as specific as possible about what data it actually wants. Tricks include first selecting 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 really wanted. 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 3 earners on the payroll, the proper way to make this query is with a correlated subquery. Table 3-1 shows the entire table returned by the SQL statement

```
select * from payroll
```

Table 3-1 Full Results Returned

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

Here 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 *3 rows, with the name and salary of the top 3 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 will obtain all the locks necessary on the various rows and tables, and will use them and release 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 above transaction might fail, due, for instance, to violating a unique key constraint, you should put in conditional SQL logic to detect any statement failure and rollback the transaction rather than commit. If, in the above example, the insert failed, most DBMSs will send back an error message about the failed insert, but will behave as if you got the message between the second and third statement, and decided to commit anyway! Microsoft SQL Server has a nice 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 which 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 user input is needed to form or complete a transaction, use optimistic locking. Briefly, optimistic locking employs timestamps and triggers (some DBMSs will generate these automatically with tables set up for it) 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 will automatically update the relevant timestamps for changed data. If any interceding update from another client has altered any of the data on which the current transaction is based, the timestamps will have changed, and the current transaction will be rejected. Most of the time, no relevant data has been changed so transactions usually succeed. When one a transaction fails, the application can refetch the updated data to present to the user to reform the transaction if desired.

Refer to your DBMS documents for a full description of this technique.

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. The trade-off is that your table may require more disk space but may run faster. Since 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.

4 Configuring WebLogic JDBC Features

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

Using Connection Pools

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

Advantages to Using Connection Pools

Connection Pools provide numerous performance and application design advantages:

- Using Connection Pools is far more efficient than creating a new connection for each client each time they need to access the database.

- You do not need to hard-code details such as the DBMS password in your application.
- You can limit the number of connections to your DBMS. This can be useful for managing licensing restrictions on the number of connections to your DBMS.
- You can change the DBMS you are using without changing your application code.

The attributes for 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 4-8. You can also use the command line; see the Web Logic Server Command-Line Interface Reference in the Administration Guide at `http://e-docs.bea.com/wls/docs61/adminguide/cli.html`.

Connection Pool Fail-Over Requirements

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.

Creating a Connection Pool at Startup

You set attributes in the in the Administration Console to create a startup (static) connection pool. The WebLogic Server opens JDBC connections to the database during the startup process and adds the connections to the pool.

Here is a list with descriptions of the connection pool attributes. For more information see "Managing JDBC Connectivity" in the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html` and the Administration Console Online Help at `http://e-docs.bea.com/wls/docs61/ConsoleHelp/jdbcconnectionpool.html`.

Connection Pool Attributes

Name

(Required) Name of the connection pool. You use the name to access a JDBC Connection from this pool.

URL

(Required) URL of the JDBC 2-tier driver for the connection between the WebLogic Server and the DBMS. You can use one of the WebLogic jDrivers or another JDBC driver that you have tested in a 2-tier environment. Check the documentation for the JDBC driver you choose to find the URL.

Driver Class Name

(Required) Full package name of the JDBC 2-tier driver class for the connection between the WebLogic Server and the DBMS. Check the documentation for the JDBC driver to find the full pathname.

Properties

(Required) The properties for connecting to the database, such as username, server, and open string for XA connections. For the database password, use the Password property. For the password in the open string, use the Open String Password attribute.

The properties are defined by, and processed by, the two-tier JDBC driver that you use. Check the documentation for the JDBC driver to find the properties required to connect to your DBMS.

Password

(Optional) The database password passed to the 2-tier JDBC driver when creating physical database connections. This value overrides any database password defined in the Properties attribute (as a name/value pair). The value is stored in an encrypted form in the config.xml file.

Open String Password

(Optional) The password is used in the open string for creating an XA physical database connection. This value overrides the password in the open string defined in the Properties attribute. The value is stored in an encrypted form in the config.xml file.

Login Delay Seconds

(Optional) Number of seconds to wait between each attempt to open a connection to the database. Some database servers can't handle multiple

requests for connections in rapid succession. This property allows you to build in a small delay to let the database server catch up.

Initial Capacity

(Optional) The initial size of the pool. If this value is unset, the default is the value you set for **CapacityIncrement.**

Maximum Capacity

(Required) The maximum size of the pool.

Capacity Increment

The size by which the pool's capacity is enlarged. **Initial Capacity** and **Capacity Increment** work somewhat like a Java Vector, which has an initial allocation (its "capacity") and is increased in increments as necessary (capacityIncrement), up to the pool **Maximum Capacity.** The default value is 1.

Allow Shrinking

(Optional) Whether this connection pool should be allowed to shrink back to its initial capacity, after expanding to meet increased demand. Set **Shrink Period** if this property is set to `true`, or it will default to 15 minutes. Note that **Allow Shrinking** is set by default to `false`, for backwards compatibility.

Shrink Period

(Optional) The number of minutes to wait before shrinking a connection pool that has incrementally increased to meet demand. The default shrink period is 15 minutes and the minimum is 1 minute.

Note: If you set a value for this attribute when AllowShrinking is set to false, WebLogic Server *ignores* the false setting and allows shrinking according to the value in ShrinkPeriodMins.

Test Table Name

(Required only if you set **Refresh Period**, **Test Reserved Connections**, or **Test Released Connections**.) The name of a table in the database that is used to test the viability of connections in the connection pool. The query `select count(*) from` **Test Table Name** is used to test a connection. The **Test Table Name** must exist and be accessible to the database user for the connection. Most database servers optimize this SQL to avoid a table scan, but it is still a good idea to set **Test Table Name** to the name of a table that is known to have few rows, or even no rows.

Refresh Period

(Optional) This property, together with the **Test Table Name** property, enables autorefresh of connections in the pools. At a specified interval, each *unused* connection in the connection pool is tested by executing a simple SQL query on the connection. If the test fails, the connection's resources are dropped and a new connection is created to replace the failed connection. The default value is 0.

To enable autorefresh, set **Refresh Period** to the number of minutes between connection test cycles—a value greater than or equal to 0. If you set an invalid **Refresh Period** value, the value defaults to 5 minutes. Set **Test Table Name** to the name of an existing database table to use for the test (required).

Also see "Notes About Refreshing Connections in a JDBC Connection Pool."

Test Reserved Connections

(Optional) When set to `true`, the WebLogic Server tests a connection after removing it from the pool and before giving it to the client. The test adds a small delay in serving the client's request for a connection from the pool, but ensures that the client receives a working connection (assuming that the DBMS is available and accessible). The Test Table Name parameter must be set to use this feature.

When using a connection pool in a MultiPool with the High Availability algorithm, you *must* set this attribute to `true` so that the MultiPool can determine when to fail over to the next connection pool in the list. See "MultiPool Fail-Over Limitations and Requirements" on page 4-18.

Test Released Connections

(Optional) When set to `true`, the WebLogic Server tests a connection before returning it to the connection pool. If all connections in the pool are already in use and a client is waiting for a connection, the client's wait will be slightly longer while the connection is tested. The Test Table Name parameter must be set to use this feature.

Notes About Refreshing Connections in a JDBC Connection Pool

When the refresh process finds a bad database connection that it cannot replace, the process stops its current cycle. It does not delete remaining broken connections from the connection pool. They remain in the connection pool until they can be replaced by new connections. This behavior was designed to avoid degrading performance by using system cycles to refresh database connections when the DBMS is inaccessible.

The refresh process cannot test or refresh connections currently being used by application code. It will only test connections that are not currently reserved. Thus a refresh cycle, even if it is able to replace any bad connections it finds, may never test all connections in the connection pool if applications are requesting connections.

Because the refresh process can only test connections not in use, it's possible that some connections will never be tested. A client will always run the risk of getting a broken connection unless `testConnsOnReserve` is enabled. In fact, even if the connection is tested before being given to an application, the connection could go bad immediately after the successful test.

Permissions

Set the permissions for creating dynamic connection pools in the Administration Console.You associate an ACL with a dynamic connection pool when you create the connection pool. The ACL and connection pool are not required to have the same name, and more than one connection pool can make use of a single ACL. If you do not specify an ACL, the "system" user is the default administrative user for the pool and any user can use a connection from the pool.

If you define an ACL for connection pools, access is restricted to *exactly* what is defined in the ACL. For example, before you have any ACLs for connection pools in your `fileRealm.properties` file, everyone has unrestricted access to all connection pools in your domain. However, if you add the following line to the file, access becomes very restricted:

```
acl.reset.weblogic.jdbc.connectionPool=Administrators
```

This line grants reset privileges to Administrators on all connection pools *and it prohibits all other actions by all other users*. By adding an ACL, file realm protection for connection pools is activated. WebLogic Server enforces the ACLs defined in `fileRealm.properties` and only allows access specifically granted in the file. If your intent in adding the ACL was to restrict resets only on connection pools, you must specifically grant privileges for other actions to everyone or to specific roles or users. For example:

```
acl.reserve.weblogic.jdbc.connectionPool=everyone
acl.shrink.weblogic.jdbc.connectionPool=everyone
acl.admin.weblogic.jdbc.connectionPool=everyone
```

Table 4-1 lists the ACLs that you can use in `fileRealm.properties` to secure connection pools.

Table 4-1 File Realm JDBC ACLs

Use this ACL. . .	To Restrict. . .
`reserve.weblogic.jdbc.connectionPool[.poolname]`	Reserving connections in a connection pool.
`reset.weblogic.jdbc.connectionPool[.poolname]`	Resetting all the connections in a connection pool by shutting down and reestablishing all allocated connections.
`shrink.weblogic.jdbc.connectionPool[.poolname]`	Shrinking the connection pool to its original size (number of connections).
`admin.weblogic.jdbc.connectionPool[.poolname]`	Enabling, disabling, and shutting down the connection pool.
`admin.weblogic.jdbc.connectionPoolcreate`	Creation of connection pools.

For information on how to modify ACLs, see Defining ACLs in the Managing Security section of the Administration Guide at `http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html`.

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 not 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. When the connection is returned to the connection pool and then reused, the client will still only get 3 rows returned, even if the table they are selecting against has 500 rows. In most

cases, there is standard (non-DBMS–specific) SQL code that can accomplish the same result and for which WebLogic Server or the JDBC driver will reset the connection. In this example, you could use `setMaxRows()` instead of `set rowcount`.

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

Creating a Connection Pool Dynamically

A JNDI-based API allows you to create a connection pool from within a Java application. With this API, you can create a connection pool in a WebLogic Server that is already running. Access to dynamic connection pools requires a JTS or Pool driver.

Dynamic pools can be temporarily disabled, which suspends communication with the database server through any connection in the pool. When a disabled pool is enabled, the state of each connection is the same as when the pool was disabled; clients can continue their database operations right where they left off.

Properties

To define a specific property for your connection pool, be sure that you duplicate the key's exact spelling and case. You pair these types (keys) along with their values, shown in the table below, in a `java.utils.Properties` object that is used when creating the pool.

Table 4-2 Connection Pool Properties

Property Type	Description	Sample Property Value
`poolName`	Required. Unique name of pool.	`myPool`
`aclName`	Required. Identifies the different access lists within `fileRealm.properties` in the server config directory. Paired name must be dynaPool.	`dynaPool`
`props`	Database connection properties; typically in the format "database login name; server network id".	`user=scott; server=ora817`

Table 4-2 Connection Pool Properties

Property Type	Description	Sample Property Value
`password`	Optional. Database password passed to the 2-tier JDBC driver when creating physical database connections. This value overrides any database password defined in props (as a name/value pair). The value is stored in an encrypted form in the config.xml.	`tiger`
`xapassword`	Optional. Password is used in the open string for creating an XA physical database connection. This value overrides the password in the open string defined in props. The value is stored in an encrypted form in the config.xml.	`secret`
`initialCapacity`	Initial number of connections in a pool. If this property is defined and a positive number > 0, WebLogic Server creates these connections at boot time. Default is 0; cannot exceed maxCapacity.	`1`
`maxCapacity`	Maximum number of connections allowed in the pool. Default is 1; if defined, maxCapacity should be =>1.	`10`
`capacityIncrement`	Number of connections that can be added at one time. Default = 0.	`1`
`allowShrinking`	Indicates whether or not the pool can shrink when connections are detected to not be in use. Default = true.	`True`
`shrinkPeriodMins`	Interval between shrinking. If `allowShrinking = True`, then default = 15 minutes.	`5`

Table 4-2 Connection Pool Properties

Property Type	Description	Sample Property Value
`driver`	Required. Name of JDBC driver. Only local (non-XA) drivers can participate.	`weblogic.jdbc.oci.Driver`
`url`	Required. URL of the JDBC driver.	`jdbc:weblogic:oracle`
`testConnectionsOnReserve`	Indicates reserved test connections. Default = False.	`true`
`testConnectionsOnRelease`	Indicates test connections when they are released. Default = False.	`true`
`testTableName`	Database table used when testing connections; must be present for tests to succeed. Required if `testConnectionsOnReserve`, `testConnectionsOnRelease`, or `refreshPeriod` is defined.	`myTestTable`
`refreshPeriod`	Interval between connection testing.	`1`
`loginDelaySecs`	Seconds between each login attempt. Default = 0.	`1`

Dynamic Connection Pool Sample Code

The following sample code shows how to create a connection pool programmatically.

Note: The following code samples cannot be used in a clustered environment. As a work-around, you can create connection pools and data sources in the Administration Console as described in the Administration Console Online Help, and target the connection pools and data sources to a cluster.

Import Packages

Import the following packages:

```
import java.util.Properties
import weblogic.common.*;
import weblogic.jdbc.common.JdbcServices;
import weblogic.jdbc.common.Pool;
```

Use JNDI to Retrieve the JdbcServices Object

The object reference allows you to access all the methods needed to create the dynamic pool. First, get an initial JNDI context to the WebLogic JNDI provider, and then look up "`weblogic.jdbc.common.JdbcServices`."

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
// URL for the WebLogic Server
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.SECURITY_PRINCIPAL, “Fred”);
env.put(Context.SECURITY_CREDENTIALS, “secret”);

Context ctx = new InitialContext(env);

// Look up weblogic.jdbc.JdbcServices
weblogic.jdbc.common.JdbcServices jdbc =
 (weblogic.jdbc.common.JdbcServices)
   ctx.lookup("weblogic.jdbc.JdbcServices");
```

Set the Properties

Set up the java.utils.properties object that defines the attributes of your pool. See Table 4-2 Connection Pool Properties in "Properties" on page 4-8

Once you have loaded `weblogic.jdbc.JdbcServices`, you pass the `weblogic.jdbc.common.JdbcServices.createPool()` method a Properties object that describes the pool. The Properties object contains the same properties you use to create a connection pool in the Administration Console, except that the "`aclName`" property is specific to dynamic connection pools.

The following example creates a connection pool named "eng2" for the DEMO Oracle database. The connections log into the database as user "SCOTT" with password "tiger." When the pool is created, one database connection is opened. A maximum of ten connections can be created on this pool. The "aclName" property specifies that the connection pool will use the "dynapool".

```
String thePoolName = “eng2”;
Properties poolProps = null;
Pool myPool = null;
weblogic.jdbc.common.Pool pool = null

poolProps = new Properties();

 // Set properties for the ConnectionPool.
poolProps.put("poolName", thePoolName);
poolProps.put("url", "jdbc:weblogic:oracle");
poolProps.put("driver", "weblogic.jdbc.oci.Driver");
poolProps.put("props", "user=scott;password=tiger;server=demo");
poolProps.put("password", "tiger");
poolProps.put("initialCapacity", "1");
poolProps.put("maxCapacity", "10");
poolProps.put("capacityIncrement", "1");
poolProps.put("aclName", "weblogic.jdbc.connectionPool.dynapool");
poolProps.put("allowShrinking", "true");
poolProps.put("shrinkPeriodMins", "5");
poolProps.put("refreshPeriod", "10");
poolProps.put("testConnectionsOnReserve", "true");
poolProps.put("testConnectionsOnRelease", "false");
poolProps.put("testTableName", "dual");
poolProps.put("loginDelaySecs", "1");
```

Create the Dynamic Pool

Create the pool by passing in the newly defined Properties object to the JdbcServices object previously retrieved from JNDI. An exception is thrown if there is a problem creating the pool, such as a new pool with the same name as an existing pool.

```
// create our pool

        try {

            myJdbc.createPool(poolProps);
        } catch (Exception e) {
            System.out.println(thePoolName
                        + " can't be created ..");
            System.exit(666);
        }
```

Retrieve the Pool Handle

Retrieve the pool handle from the newly created pool.You use the pool handle to manipulate the pool during the course of the application.

```
 weblogic.jdbc.common.Pool myPool = null;

 // get our pool, we would like to do something with it...

try {

    theNewPool = myJdbc.getPool(thePoolName);

} catch (Exception e) {
   System.out.println("Cannot retrieve pool: "
                  +  thePoolName);
    System.exit(666);

 }
```

Managing Connection Pools

The `weblogic.jdbc.common.Pool` and `weblogic.jdbc.common.JdbcServices` interfaces provide methods to manage connection pools and obtain information about them. Methods are provided for:

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

Retrieving Information About a Pool

`weblogic.jdbc.common.JdbcServices.poolExists()`

`weblogic.jdbc.common.Pool.getProperties()`

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

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

Disabling a Connection Pool

`weblogic.jdbc.common.Pool.disableDroppingUsers()`

`weblogic.jdbc.common.Pool.disableFreezingUsers()`

`weblogic.jdbc.common.pool.enable()`

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

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

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

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

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

Shrinking a Connection Pool

`weblogic.jdbc.common.Pool.shrink()`

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

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

Shutting Down a Connection Pool

`weblogic.jdbc.common.Pool.shutdownSoft()`

`weblogic.jdbc.common.Pool.shutdownHard()`

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

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

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

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

Resetting a Pool

`weblogic.jdbc.common.Pool.reset()`

You can configure a connection pool to test its connections either periodically, or every time a connection is reserved or released. Allowing the WebLogic Server to automatically maintain the integrity of pool connections should prevent most DBMS

connection problems. In addition, WebLogic provides methods you can call from an application to refresh all connections in the pool or a single connection you have reserved from the pool.

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

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

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

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

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

- The `reset()` method from the `weblogic.common.Pool` interface in your client application.

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

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

  b. Catch the `SQLException`.

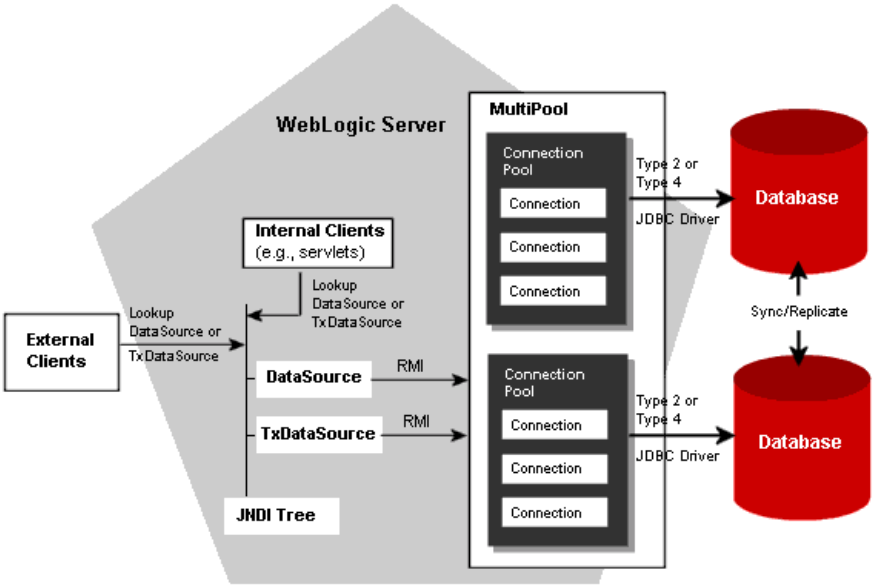
  c. Call the `reset()` method in the catch block.

Using MultiPools

A MultiPool is a “pool of pools.” MultiPools contain a configurable algorithm for determining the connection pool from which a connection is returned to an application: either high availability *or* connection pool load balancing.

MultiPools differ from connection pools in that all the connections in a particular *connection pool* are created identically with a single database, single user, and the same connection attributes. However, the connection pools within a *MultiPool* may be associated with different users or DBMSs.

Figure 4-1 MultiPool Architecture



Note that although a Multipool can return connections from multiple databases or with different users, WebLogic Server does not provide any means to integrate or handle the contents of disparate databases. Your application or DBMS environment must handle the synchronization or data integration so that your application will work transparently and successfully when it receives a connection from any of the underlying connection pools.

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 the first pool loses connectivity to the database, then a connection is sought sequentially from the next pool on the list.

Notes: You must set `TestConnectionsOnReserve=true` for the connection pools within the MultiPool so that the MultiPool can determine when to fail over to the next connection pool in the list.

If all connections in a 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. This is by design so that you can set the capacity for a connection pool. MultiPool fail-over takes effect only if loss of database connectivity has occurred. To avoid this situation, you should increase the maximum number of connections in the connection pool.

Load Balancing

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

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.

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.

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.

Guidelines to Setting Wait for Connection Times

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

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

Messages and Error Conditions

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

Exceptions

Exceptions are posted to the JDBC log under these circumstances:

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

Capacity Issues

In a high availability scenario, the fact that the first pool in the list is busy does not trigger an attempt to get a connection from the next pool in the list.

Configuring and Using DataSources

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

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

WebLogic Server supports two types of DataSource objects:

- DataSources (for local transactions only)
- TxDataSources (for distributed transactions)

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

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

For more information about when to use a TxDataSource and how to configure a TxDataSource, see JDBC Configuration Guidelines for Connection Pools, MultiPools, and DataSources in the *Administration Guide* at `http://e-docs.bea.com/wls/docs61/adminguide/jdbc.html#jdbc002`.

If you want applications to use a DataSource to get a database connection from a connection pool (the preferred method), you should define the DataSource in the Administration Console before running your application. For instructions to create a DataSource, see the Administration Console Online Help at `http://e-docs.bea.com/wls/docs61/ConsoleHelp/jdbcdatasource.html`. For instructions to create a TxDataSource, see the Administration Console Online Help at `http://e-docs.bea.com/wls/docs61/ConsoleHelp/jdbctxdatasource.html`.

Importing Packages to Access DataSource Objects

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

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

Obtaining a Client Connection Using a DataSource

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

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

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

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

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

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

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

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

Note: The code above uses one of several available procedures for obtaining a JNDI context. For more information on JNDI, see Programming WebLogic JNDI at `http://e-docs.bea.com/wls/docs61/jndi/index.html`.

Code Examples

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

CHAPTER

5 Using WebLogic Multitier JDBC Drivers

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

Overview of WebLogic Multitier Drivers

You can access multitier drivers in the following ways:

- New applications. BEA recommends using DataSource objects for new applications. DataSource objects, along with the JNDI, provide access to connection pools for database connectivity. Each data source requires a separate DataSource object, which may be implemented as a DataSource class that supports either connection pooling, or distributed transactions. For more information, see "Configuring WebLogic JDBC Features" on page 4-1.
- Existing applications. For existing applications that use the JDBC 1.x API, refer to the following sections.

Using the WebLogic RMI Driver

The WebLogic RMI driver is a multitier, Type 3, JDBC driver that runs in WebLogic Server, used with:

- Two-tier JDBC drivers, including drivers in the WebLogic jDriver family, to provide database access for local transactions
- Two-tier JDBC XA drivers, including the WebLogic jDriver for Oracle/XA, for distributed transactions. For additional information, see "Oracle Thin Driver Extensions" on page 6-15, regarding information about using Oracle Thin Driver 8.1.7.

The BEA WebLogic RMI driver operates with WebLogic Server. The DBMS connection is made by means of the WebLogic Server, a *DataSource* object, and a *connection pool* operating in WebLogic Server.

The *DataSource* object provides access to RMI driver connections. The connection parameters are set in the Administration Console. This connection pool is in turn configured for two-tier JDBC access to a DBMS.

RMI driver clients make their connection to the DBMS by looking up this DataSource object. This lookup is accomplished by using a Java Naming and Directory Service (JNDI) lookup, or by directly calling the 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 in the last release) and the Pool driver, and uses the Java standard Remote Method Invocation (RMI) to connect to WebLogic Server rather than the proprietary t3 protocol.

Since 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.

Limitations When Using the WebLogic RMI Driver

Please be aware of the following:

- You cannot use Batch updates (`addBatch()`) with the `callableStatement or preparedStatement` SQL statements when using the RMI driver in conjunction with the WebLogic jDriver for Oracle or compliant third-party 2-Tier drivers.

Setting Up WebLogic Server to Use the WebLogic RMI Driver

RMI drivers are accessible only through DataSource objects, which are created in the Administration Console.

Setting Up the Client to Use the WebLogic Server

The following code samples shows how to obtain and use the connection.

Import the Following Packages

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

Obtain the Client Connection

WebLogic JDBC/RMI client obtains its connection to a DBMS from the DataSource object that was 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, the WebLogic Server performs the JNDI look up on behalf of the client.

Using a JNDI Lookup to Obtain the Connection

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

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

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

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

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

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

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

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

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI look up. For more information, see *Programming WebLogic JNDI* at http://e-docs.bea.com/wls/docs61/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 the Connection

You can also access the 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 fetches a single row from the DBMS and transmits them 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.

Important Limitations to Using 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 will only perform row caching on if the result set type is both TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.
- Certain data types in a result set may disable caching for that result set. These include the following:
  - LONGVARCHAR/LONGVARBINARY
  - NULL
  - BLOB/CLOB
  - ARRAY
  - REF
  - STRUCT
  - JAVA_OBJECT
- Certain ResultSet methods are not supported if row caching is enabled and active for that result set. Most pertain to streaming data, scrollable result sets or data types not supported for row caching. These include the following:
  - `getAsciiStream()`
  - `getUnicodeStream()`
  - `getBinaryStream()`
  - `getCharacterStream()`
  - `isBeforeLast()`
  - `isAfterLast()`
  - `isFirst()`

- isLast()
- getRow()
- getObject (Map)
- getRef()
- getBlob()/getClob()
- getArray()
- getDate()
- getTime()
- getTimestamp()

Using the WebLogic JTS Driver

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

Once a transaction is begun, all of the database operations in a execute thread that get their connection from the *same connection pool* will share the *same connection* from that pool. These operations may 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 will be 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 will 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.

When you select Enable Two Phase Commit (`enableTwoPhaseCommit = true`) for a Tx Data Source with a non-XA JDBC driver, WebLogic Server uses the JTS driver internally to enable the non-XA resource to emulate two-phase commit (2PC) and participate in global transactions. For more information about enabling non-XA resources to participate in global transactions and how the JTS driver is used, see "Configuring Non-XA JDBC Drivers for Distributed Transactions" in the *Administration Guide*.

Implementing with 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 Connection Pools in Managing JDBC Connectivity in *Administration Guide*.

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. This class can be looked up in the Java Naming and Directory Service (JNDI). The `UserTransaction` class controls the transaction on the current execute thread. Note that this class does not represent the transaction itself. The actual context for the transaction is associated with the current execute thread.

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

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

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

```
ctx = new InitialContext(env);

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

3. Start a transaction on the current thread:

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

4. Load the JTS driver:

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

5. Get a connection from the connection pool:

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

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

6. Execute your database operations. These operations may be made by any service that uses a database connection. These include EJB, JMS, or standard JDBC statements. If these operations use the JTS driver to access the same connection pool as the transaction begun in step 3, they will participate in that transaction.

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

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

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

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

9. Complete the transaction by either committing the transaction or rolling it back. The JTS driver will commit all the transactions on all connection objects in the current thread and return 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 on using the Pool driver, see Accessing Databases in Programming Tasks in *Programming WebLogic HTTP Servlets*.

6 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
- Are EJB accessible; can implement transaction calls in JDBC

This section describes how to set up and use the following third-party two-tier, Type 4 drivers with WebLogic Server:

- Oracle Thin Driver

- Sybase jConnect Driver

The Sybase jConnect Driver (versions 4.2/5.2 and 5.5) and Oracle Thin Driver (version 9.2.0) are bundled with your WebLogic Server distribution; the `weblogic.jar` file contains the Oracle Thin Driver and Sybase jConnect classes. Additional information about these Oracle and Sybase drivers is available at their respective Web sites.

Note: With the release of WebLogic Server 6.1 Service Pack 4, the version of the Oracle Thin Driver bundled in weblogic.jar is version 9.2.0. Previous releases of WebLogic Server 6.1 included the 8.1.7 version of the Oracle Thin Driver.

Limitations

Please be aware of the following:

- You cannot use Batch updates (`addBatch()`) with the `callableStatement or preparedStatement` SQL statements when using the RMI driver in conjunction with 2-Tier drivers.

Setting the Environment for Your Third-Party Driver

If you use a third-party JDBC driver other than the Oracle Thin Driver or Sybase jConnect Driver included in `weblogic.jar`, you must add the path for the JDBC driver classes to your `CLASSPATH`. The following topics describe how to set your `CLASSPATH` for Windows and UNIX when using a third-party JDBC driver.

CLASSPATH for Third-Party Driver on Windows

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

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

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

CLASSPATH for Third-Party Driver on UNIX

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

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

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

Updating Oracle Thin Driver

To update the Oracle Thin Driver bundled with WebLogic Server, you must add the path for the new driver classes to your `CLASSPATH` in front of the path to `weblogic.jar`. For example:

```
set CLASSPATH=%ORACLE_HOME%\jdbc\lib\classes12.zip;
%WL_HOME%\lib\weblogic.jar;%CLASSPATH% (Windows)
```

Or

```
export CLASSPATH=$ORACLE_HOME/jdbc/lib/classes12.zip:
$WL_HOME/lib/weblogic.jar:$CLASSPATH (UNIX)
```

Use this procedure for updates to Oracle Thin Driver version 9.2.0 bundled in weblogic.jar or to use a new version of the driver.

The Oracle Thin Driver is included with the Oracle DBMS software. You can also download driver updates from the Oracle Web site at http://otn.oracle.com/software/content.html.

Using the Oracle 10g Thin Driver

The Oracle 10g Thin driver is installed with WebLogic Server 6.1 Service Pack 7 (and later service packs) in the *`WL_HOME`*`\lib\oracle\10g` folder. BEA provides the 10g driver so that you can optionally use it with WebLogic Server. However, the Oracle 9.2.0 Thin driver remains the default version of the driver. The 9.2.0 version is included in `weblogic.jar` and is included in your `CLASSPATH` when you run the server.

To use the Oracle10g Thin driver, you must add the driver classes to your `CLASSPATH` in front of `weblogic.jar`, preferably in your scripts to start WebLogic Server. For example, in the *`WL_HOME`*`\config\mydomain\startWeblogic.cmd` (`.sh` on UNIX), change the following line:

```
set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar
```

To this:

```
set CLASSPATH=.;WL_HOME\lib\oracle\10g\classes12.zip;
.\lib\weblogic_sp.jar;.\lib\weblogic.jar
```

Note: Line break added for readability.

In this example, *`WL_HOME`* is the directory in which WebLogic Server is installed, typically `C:\bea\wlserver6.1`.

Note: Oracle removed some methods and classes from the 10g Thin driver that were in previous versions of the driver. If you use any of the extension methods that were removed, you will see errors in your application.

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` package and 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.

Updating Sybase jConnect Driver

To update the Sybase jConnect Driver bundled with WebLogic Server, you must add the path for the jConnect driver to your `CLASSPATH` in front of the path to `weblogic.jar`. For example:

```
set CLASSPATH=%SYBASE_HOME%\jConnect-5_5\classes\jconn2.jar;
%WL_HOME%\lib\weblogic.jar;%CLASSPATH% (Windows)
```

Or

```
export CLASSPATH=$SYBASE_HOME/jConnect-5_5/classes/jconn2.jar:
$WL_HOME/lib/weblogic.jar:$CLASSPATH (UNIX)
```

The Sybase jConnect Driver (`jConnect.jar`) is included with the Sybase DBMS software. You can also download driver updates from the Sybase Web site at http://www.sybase.com/products/eaimiddleware/jconnectforjdbc.

Installing and Using the IBM Informix JDBC Driver

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

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

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

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

   `WL_HOME` is the folder where you installed WebLogic Server, typically `c:\bea\wlserver6.1`.

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

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

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

Connection Pool Attributes when using the IBM Informix JDBC Driver

Use the attributes as described in Table 6-1 and Table 6-2 when creating a connection pool that uses the IBM Informix JDBC driver.

Table 6-1 Non-XA Connection Pool Attributes Using the Informix JDBC Driver

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

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

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

Table 6-2 XA Connection Pool Attributes Using the Informix JDBC Driver

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

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

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

```
<JDBCConnectionPool CapacityIncrement="2"
    DriverName="com.informix.jdbcx.IfxXADataSource"
```

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

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

Programming Notes for the IBM Informix JDBC Driver

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

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

Getting a Connection with Your Third-Party Driver

The following topics describe two ways to get a connection using a third-party, Type 4 driver, such as the Oracle Thin Driver and Sybase jConnect Driver. BEA recommends you use connection pools, data sources, and JNDI Lookup to establish your connection. As an alternative, you can get a simple connection directly between the Java client and the database.

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 Managing JDBC Connectivity in the *Administration Guide* for information on how to use the Administration Console to:

- Create a JDBC Connection Pool
- Create a JDBC DataSource

Using a JNDI Lookup to Obtain the Connection

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

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

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

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

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

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

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

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

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

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI look up. For more information, see Programming WebLogic JNDI at `http://e-docs.bea.com/wls/docs61/jndi/index.html`.

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

Getting a Physical Connection from a Connection Pool

When you get a connection from a connection pool, WebLogic Server provides a logical connection rather than a physical connection so that WebLogic Server can manage the connection with the connection pool. This is necessary to enable connection pool features and to maintain the quality of connections provided to applications. In some cases, you may want to use a physical connection, such as if you need to pass the connection to a DBMS vendor-specific method that requires the vendor's connection class. WebLogic Server includes the `getVendorConnection()` method in the `weblogic.jdbc.extensions.WLConnection` interface that you can use to get the underlying physical connection from a logical connection. See the WebLogic Javadocs at `http://e-docs.bea.com/wls/docs61/javadocs/weblogic/jdbc/extensions/WLConnection.html`.

Note: BEA strongly discourages using a physical connection instead of a logical connection from a connection pool. See "Limitations for Using a Physical Connection" on page 6-13.

You should only use the physical database connection for vendor-specific needs. Your code should continue to make most JDBC calls to the logical connection.

When you are finished with the connection, you should close the logical connection. Do not close the physical connection in your code.

Whenever a physical database connection is exposed to application code, the connection pool cannot guarantee that the next user of that connection will be the only user with access to it. Therefore, when the logical connection is closed, WebLogic Server returns the logical connection to the connection pool, but discards the underlying physical connection and opens a new physical connection for the logical connection in the pool. This is safe, but it is also slow. It is possible that every request to the connection pool will entail making a new database connection.

Code Sample for Getting a Physical Connection

To get a physical database connection, you first get a connection from a connection pool as described in "Using a JNDI Lookup to Obtain the Connection" on page 6-9, then do one of the following:

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

For example:

```
//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)).ge
tVendorConnection());

  // 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){}
  }
}
```

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 be used in server-side code only.
- When you use a physical connection, you lose all of the connection management benefits that WebLogic Server offers, including error handling, statement caching, and so forth.
- 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.
- The connection is not reused. When you close the connection, the physical connection is closed and the connection pool creates a new connection to replace the one passed as a physical connection. Because the connection is not reused, there is a performance loss when using a physical connection because of the following:
  - The physical connection 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.

Obtaining a Direct (Non-pooled) JDBC Connection

This simple example shows you how to establish a connection directly between the your Java code running on WebLogic Server and the database. Use `driver.connect()` to set your direct connection. Do not use DriverManager to get a JDBC connection because the DriverManager methods are overly synchronized for a multi-threaded application, and can cause WebLogic Server to become single-threaded or to lock up.

The following example shows how to get a direct connection using a third-party driver.

Obtaining a Direct Connection Using the Oracle Thin Driver

The following example shows how to set a direct connection using the Oracle Thin Driver:

- Instantiate the driver:

```
// ThinDriver driver
driver = (Driver)Class.forName
   ("oracle.jdbc.driver.OracleDriver").newInstance();

Properties props = new Properties();
props.put("user", "scott");
props.put("password", "tiger");
```

- Make the connection:

```
// Thin driver connection
con = driver.connect
   ("jdbc:oracle:thin:@myHost.mydomain.com:1521:DEMO", props);
```

Obtaining a Direct Connection Using the Sybase jConnect Driver

The following example shows how to set a direct connection using the Sybase jConnect Driver:

- Instantiate the driver:

```
// Sybase jConnect driver
driver = (Driver)Class.forName
   (“com.sybase.jdbc.SybDriver”).newInstance()
```

```
Properties props = new Properties();
props.put("user", "scott");
props.put("password", "tiger");
```

- Make the connection:

```
// Sybase jConnect
con = driver.connect
   ("jdbc:sybase:Tds:myDB@myhost:myport), props);
```

Oracle Thin Driver Extensions

BEA supports the following extensions to the Oracle Thin Driver for use with the RMI, JTS and Pool drivers:

Oracle Standard Extensions

- `OracleConnection`
- `OracleStatement`
- `OracleResultSet`
- `OraclePreparedStatement`
- `OracleCallableStatement`

Oracle Blobs and Clobs

- `OracleThinBlob`
- `OracleThinClob`

The following sections provide code samples for Oracle extensions and tables of supported methods. For more information, please refer to the Oracle documentation.

Sample Code for Accessing Oracle Extensions to JDBC Interfaces

The following code examples show how to access the WebLogic Oracle extensions to standard JDBC interfaces. The following example uses the OracleConnection and OracleStatement extensions. You can use the syntax of this example for the OracleResultSet, OraclePreparedStatement and OracleCallableStatement interfaces, when using methods supported by WebLogic Server. For supported methods, see "Tables of Oracle Interfaces" on page 6-20.

For examples showing how to access the OracleThinBlob and OracleThinClob interfaces, see "Oracle Blob/Clob Extensions and Supported Methods" on page 6-27.

Import Packages to Access Oracle Extensions

Import the Oracle interfaces used in this example. The OracleConnection and Oracle Statement interfaces are counterparts to oracle.jdbc.OracleConnection and oracle.jdbc.OracleStatement and can be used in the same way as the Oracle interfaces when using the methods supported by WebLogic Server.

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

Establish the Connection

Establish the connection using JNDI, DataSource and connection pool objects. For information, see "Using a JNDI Lookup to Obtain the Connection" on page 6-9.

```
// 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();
```

Retrieve the Default Row Prefetch Value

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

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

int default_prefetch =
          ((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.
      ((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;
    }
```

Sample Code for Accessing Oracle Blob/Clob Interfaces

This section contains sample code that demonstrates how to access the OracleThinBlob interface. You can use the syntax of this example for the OracleThinBlob interface, when using methods supported by WebLogic Server. See “Tables of Oracle Interfaces” on page 6-20.

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`.

Import Packages to Access Blob and Clob Extensions

Include the WebLogic Server Oracle Blob with the imported classes:

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

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.

Tables of Oracle Interfaces

The following tables list the Oracle interfaces.

Oracle Extensions and Supported Methods

The following tables describe the Oracle interfaces and supported methods you use with the Oracle Thin Driver to extend `java.sql.*` interfaces. See "Oracle Blob/Clob Extensions and Supported Methods" on page 6-27 for the Blob/Clob interfaces.

Table 6-3 OracleConnection Interface

Extends	Method Signature

`OracleConnection extends java.sql.Connection`	`boolean` **`getAutoClose()`** `throws java.sql.SQLException;`
	
	`void` **`setAutoClose(boolean on)`** `throws java.sql.SQLException;`
	
	`String` **`getDatabaseProductVersion()`** `throws java.sql.SQLException;`
	
	`String` **`getProtocolType()`** `throws java.sql.SQLException;`
	
	`String` **`getURL()`** `throws java.sql.SQLException;`
	
	`String` **`getUserName(`**`) throws java.sql.SQLException;`
	
	`boolean` **`getBigEndian()`** `throws java.sql.SQLException;`
	
	`boolean` **`getDefaultAutoRefetch()`** `throws java.sql.SQLException;`
	
	`boolean` **`getIncludeSynonyms()`** `throws java.sql.SQLException;`
	
	`boolean` **`getRemarksReporting()`** `throws java.sql.SQLException;`
	
	`boolean` **`getReportRemarks()`** `throws java.sql.SQLException;`
	
	`boolean` **`getRestrictGetTables()`** `throws java.sql.SQLException;`
	
	`boolean` **`getUsingXAFlag()`** `throws java.sql.SQLException;`
	
	`boolean` **`getXAErrorFlag()`** `throws java.sql.SQLException;`

Table 6-3 OracleConnection Interface

Extends	Method Signature

OracleConnection extends java.sql.Connection (continued)	byte[] **getFDO(boolean b)** throws java.sql.SQLException;
	
	int **getDefaultExecuteBatch()** throws java.sql.SQLException;
	
	int **getDefaultRowPrefetch()** throws java.sql.SQLException;
	
	int **getStmtCacheSize()** throws java.sql.SQLException;
	
	java.util.Properties **getDBAccessProperties()** throws java.sql.SQLException;
	
	short **getDbCsId()** throws java.sql.SQLException;
	
	short **getJdbcCsId()** throws java.sql.SQLException;
	
	short **getStructAttrCsId()** throws java.sql.SQLException;
	
	short **getVersionNumber()** throws java.sql.SQLException;
	
	void **archive(int i, int j, String s)** throws java.sql.SQLException;
	
	void **close_statements()** throws java.sql.SQLException;
	
	void **initUserName()** throws java.sql.SQLException;
	
	void **logicalClose()** throws java.sql.SQLException;
	
	void **needLine()** throws java.sql.SQLException;
	
	void **printState()** throws java.sql.SQLException;
	
	void **registerSQLType(String s, String t)** throws java.sql.SQLException;
	
	void **releaseLine()** throws java.sql.SQLException;

Table 6-3 OracleConnection Interface

Extends	Method Signature

OracleConnection extends java.sql.Connection (continued)	void **removeAllDescriptor()** throws java.sql.SQLException;
	
	//this is Sun's spelling
	
	void **removeDecriptor(String s)** throws java.sql.SQLException;
	
	void **setDefaultAutoRefetch(boolean b)** throws java.sql.SQLException;
	
	void **setDefaultExecuteBatch(int i)** throws java.sql.SQLException;
	
	void **setDefaultRowPrefetch(int i)** throws java.sql.SQLException;
	
	void **setFDO(byte[] b)** throws java.sql.SQLException;
	
	void **setIncludeSynonyms(boolean b)** throws java.sql.SQLException;
	
	void **setPhysicalStatus(boolean b)** throws java.sql.SQLException;
	
	void **setRemarksReporting(boolean b)** throws java.sql.SQLException;
	
	void **setRestrictGetTables(boolean b)** throws java.sql.SQLException;
	
	void **setStmtCacheSize(int i)** throws java.sql.SQLException;
	
	void **setStmtCacheSize(int i, boolean b)** throws java.sql.SQLException;
	
	void **setUsingXAFlag(boolean b)** throws java.sql.SQLException;
	
	void **setXAErrorFlag(boolean b)** throws java.sql.SQLException;
	
	void **shutdown(int i)** throws java.sql.SQLException;
	
	void **startup(String s, int i)** throws java.sql.SQLException;

Note: The following method was removed in Service Pack 04:

- `isCompatibleTo816()`

Table 6-4 OracleStatement Interface

Extends	Method Signature
`OracleStatement extends java.sql.statement`	`String` **`getOriginalSql()`** `throws java.sql.SQLException;` `String` **`getRevisedSql()`** `throws java.sql.SQLException;` `boolean` **`getAutoRefetch()`** `throws java.sql.SQLException;` `boolean` **`is_value_null(boolean b, int i)`** `throws java.sql.SQLException;` `byte ge`**`tSqlKind()`** `throws java.sql.SQLException;` `int` **`creationState()`** `throws java.sql.SQLException;` `int` **`getRowPrefetch()`** `throws java.sql.SQLException;` `int` **`sendBatch()`** `throws java.sql.SQLException;` `void` **`clearDefines()`** `throws java.sql.SQLException;` `void` **`defineColumnType(int i, int j)`** `throws java.sql.SQLException;` `void` **`defineColumnType(int i, int j, String s)`** `throws java.sql.SQLException;`

Table 6-4 OracleStatement Interface

Extends	Method Signature

OracleStatement extends java.sql.statement (continued)	void **defineColumnType(int i, int j, int k)** throws java.sql.SQLException;
	
	void **describe()** throws java.sql.SQLException;
	
	void **notify_close_rset()** throws java.sql.SQLException;
	
	void **setAutoRefetch(boolean b)** throws java.sql.SQLException;
	
	void **setRowPrefetch(int i)** throws java.sql.SQLException;

Note: The following methods were removed in Service Pack 04:

- getWaitOption()
- setWaitOption(int i)
- setAutoRollback(int i)
- getAutoRollback()

Table 6-5 OracleResultSet Interface

Extends	Method Signature

OracleResultSet extends java.sql.ResultSet	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;

Note: The following method was removed in Service Pack 04:

- `getCURSOR(String s)`

.

Table 6-6 OracleCallableStatement Interface

Extends	Method Signature

`OracleCallableStatement extends java.sql. CallableStatement`	`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 6-7 OraclePreparedStatement Interface

Extends	Method Signature

OraclePreparedStatement extends OracleStatement and java.sql. PreparedStatement	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;

Oracle Blob/Clob Extensions and Supported Methods

The following tables list the extensions to the `java.sql.*` interfaces.

Table 6-8 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 6-9 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;

7 Using dbKona

The following sections describe the dbKona classes that provide high-level database connectivity to Java applications:

Introduction to dbKona

The dbKona classes provide a set of high-level database connectivity objects that give Java applications and applets access to databases. dbKona sits on top of the JDBC API and works with the WebLogic JDBC drivers, or with any other JDBC-compliant driver.

The dbKona classes provides a higher level of abstraction than JDBC, which deals with low-level details of managing data. The dbKona classes offer objects that allow the programmer to view and modify database data in a high-level, vendor-independent way. A Java application that uses dbKona objects does not need vendor-specific knowledge about DBMS table structure or field types to retrieve, insert, modify, delete, or otherwise use data from a database.

dbKona in a Multitier Configuration

dbKona may also be used in a multitier JDBC implementation consisting of WebLogic Server and a multitier driver; this configuration requires no client-side libraries. In a multitier configuration,WebLogic JDBC acts as an access method to the WebLogic multitier framework. WebLogic uses a single JDBC driver, for example, WebLogic jDriver for Oracle, to communicate from the WebLogic Server to the DBMS.

dbKona is a natural choice for writing database access programs in a multitier environment, since with its objects you may write database applications that are completely vendor independent. dbKona and WebLogic's multitier framework is particularly suited for applications that want to retrieve data from several heterogeneous databases for transparent presentation to the user.

For more information on WebLogic and the WebLogic JDBC Server, see Programming WebLogic JDBC at http://e-docs.bea.com/wls/docs61jdbc/index.html.

How dbKona and a JDBC Driver Interact

dbKona depends upon a JDBC driver to provide and maintain a connection to a DBMS. In order to use dbKona, you must have installed a JDBC driver.

- If you are using the WebLogic jDriver for Oracle native JDBC driver, you should install the appropriate WebLogic-supplied .dll, .sl, or .so for your operating system, as described in Installing and Using WebLogic jDriver for Oracle at http://e-docs.bea.com/wls/docs61/oracle/install_jdbc.html.
- If you are using a non-WebLogic JDBC driver, you should refer to the documentation for that JDBC driver.

JavaSoft's JDBC is a set of interfaces that BEA has implemented to create its jDriver JDBC drivers. BEA's JDBC drivers are JDBC implementations of database-specific drivers for Oracle, Informix, and Microsoft SQL Server. Using database-specific drivers with dbKona offers the programmer access to all of the functionality of each specific database, as well as improved performance.

Although the underlying foundation of dbKona uses JDBC for database transactions, dbKona provides the programmer with higher-level, more convenient access to the database.

How dbKona and WebLogic Events Can interact

The dbKona package contains some “eventful” classes that send and receive events (within WebLogic), using WebLogic events when data is updated locally or in the DBMS.

The dbKona Architecture

dbKona uses a high level of abstraction to describe and manipulate data that resides in a database. Classes in dbKona create and manage objects that retrieve and modify data. An application can use dbKona objects in a consistent way without any knowledge of how a particular vendor stores or processes data.

At the core of dbKona’s architecture is the concept of a `DataSet`. A `DataSet` contains the results of a query. `DataSets` allow client-side management of query results. The programmer can control the entire query result rather than dealing with a single record at a time.

A `DataSet` contains `Records`, and each `Record` contains one or more `Value` objects. A `Record` is comparable to a database row, and a `Value` can be compared to a database cell. `Value` objects “know” their internal data type as stored in the DBMS, but the programmer can treat `Value` objects in a consistent way without having to worry about vendor-specific internal data types.

Methods from the `DataSet class` (and its subclasses `TableDataSet` and `QueryDataSet`) provide a high-level, flexible way to navigate through and manipulate the results of a query. Changes made to a `TableDataSet` can be saved to the DBMS; dbKona maintains knowledge of which records have changed and makes a selective save, which reduces network traffic and DBMS overhead.

dbKona also uses other objects, like `SelectStmt` and `KeyDef` to shield the programmer from vendor-specific SQL. By using methods in these class, the programmer can have dbKona construct the appropriate SQL, which reduces syntax errors and does not require a knowledge of vendor-specific SQL. On the other hand, dbKona also allows the programmer to pass SQL to the DBMS if desired.

The dbKona API

The following sections describe the dbKona API.

The dbKona API Reference

Package weblogic.db.jdbc
Package weblogic.db.jdbc.oracle (Oracle-specific extensions)

```
Class java.lang.Object
  Class weblogic.db.jdbc.Column
   (implements weblogic.common.internal.Serializable)
  Class weblogic.db.jdbc.DataSet
   (implements weblogic.common.internal.Serializable)
    Class weblogic.db.jdbc.QueryDataSet
    Class weblogic.db.jdbc.TableDataSet
           Class weblogic.db.jdbc.EventfulTableDataSet
       (implements weblogic.event.actions.ActionDef)
    Class weblogic.db.jdbc.Enums
    Class weblogic.db.jdbc.KeyDef
    Class weblogic.db.jdbc.Record
       Class weblogic.db.jdbc.EventfulRecord
        (implements weblogic.common.internal.Serializable)
    Class weblogic.db.jdbc.Schema
     (implements weblogic.common.internal.Serializable)
    Class weblogic.db.jdbc.SelectStmt
    Class weblogic.db.jdbc.oracle.Sequence
    Class java.lang.Throwable
       Class java.lang.Exception
           Class weblogic.db.jdbc.DataSetException

    Class weblogic.db.jdbc.Value
```

The dbKona Objects and Their Classes

Objects in dbKona fall into three categories:

- *Data container objects* hold data retrieved from or bound for a database, or they contain other objects that hold data. Data container objects are always associated

with a set of data description objects and a set of session objects. `TableDataSet` and `Record` objects are examples of data container objects.

- Data description objects contain the metadata about data objects, that is, a description of how the data is structured and typed, and parameters for its retrieval from the remote DBMS. Every data object or its container is associated with a set of data description objects. `Schema` and `SelectStmt` objects are examples data description objects.
- *Miscellaneous objects* store information about errors, provide symbolic constants, etc.

These broad categories of objects depend upon each other in application building. In a general way, every data object has a set of descriptive objects associated with it.

Data Container Objects in dbKona

There are three basic objects that act as data containers: a `DataSet` (or one of its subclasses, `QueryDataSet` or `TableDataSet`) contains `Records`. A `Record` contains `Values`.

- `DataSet`
  - `QueryDataset`
  - `TableDataSet`
  - `EventfulTableDataSet` (Deprecated)
- Record
  - Value

DataSet

The dbKona package uses the concept of a `DataSet` to cache records retrieved from a DBMS server. It is roughly equivalent to a table in SQL. The `DataSet` class has two subclasses, `QueryDataSet` and `TableDataSet`.

In the multitier model using the WebLogic Server, DataSets can be saved (cached) on the WebLogic Server.

- A `DataSet` is constructed as a `QueryDataSet` or a `TableDataSet` to hold the results of a query or a stored procedure.

- A DataSet's retrieval parameters are defined by a SQL statement, or by the dbKona abstraction for SQL statements, a SelectStmt object.
- A Dataset is populated with Records, which contain Values. Records that are accessible by index position (0-origined).
- A DataSet is described by and bound to a Schema, which stores information in its attributes, like column name, data type, size, and order of each database column represented in the DataSet. Column names in a Schema are accessible by index position (1-origined).

The DataSet class (see weblogic.db.jdbc.DataSet) is the abstract parent class for QueryDataSet and TableDataSet.

QueryDataSet

A QueryDataSet makes the results of an SQL query available as a collection of Records that are accessible by index position (0-origined). Unlike the case with a TableDataSet, changes and additions to a QueryDataSet cannot be saved into the database.

There are two functional differences between a QueryDataSet and a TableDataSet. First, changes made to a TableDataSet can be saved to a database; you can make changes to Records in a QueryDataSet, but those changes cannot be saved. Second, you can retrieve data into a QueryDataSet from more than one table.

- A QueryDataSet is constructed in the context of a java.sql.Connection or with a java.sql.ResultSet; that is, you pass the Connection object as an argument to the QueryDataSet constructor. A QueryDataSet's data retrieval is specified by a SQL query and/or by a SelectStmt object.
- A QueryDataSet is populated with Records (accessible by 0-origined index), which contain Values (accessible by 1-origined index).
- A QueryDataSet is described by a Schema, which stores information about the QueryDataSet's attributes. Attributes include name, data type, size, and order of each database column represented in the QueryDataSet.

The QueryDataSet class (see weblogic.db.jdbc.QueryDataSet) has methods for constructing, saving, and retrieving a QueryDataSet. You can specify *any* SQL for a QueryDataSet, including SQL for joins. The superclass DataSet contains methods for managing record caching details.

TableDataSet

The functional difference between a `TableDataSet` and a `QueryDataSet` is that changes made to a `TableDataSet` can be saved to a database. With a `TableDataSet`, you can update values in `Records`, add new `Records`, and mark `Records` for deletion; finally, you can save changes to a database, using the `save()` methods in either the `TableDataSet` class to save an entire `TableDataSet`, or in the `Record` class to save a single record. Additionally, the data retrieved into a `TableDataSet` is, by definition, from a single database table; you cannot perform joins on database tables to retrieve data for a `TableDataSet`.

If you intend to save updates or deletes to a database, you must construct the `TableDataSet` with a `KeyDef` object that specifies a unique key for forming the `WHERE` clauses in an `UPDATE` or `DELETE` statement. A `KeyDef` is not necessary if only inserts take place, since an insert operation does not require a `WHERE` clause. The `KeyDef` key must not contain columns that are filled or altered by the `DBMS`, since dbKona must have a known value for the key column to construct a correct `WHERE` clause.

You can also qualify a `TableDataSet` with an arbitrary string that is used to construct the tail of the SQL statement. When you are using dbKona with an Oracle database, for example, you can qualify the `TableDataSet` with the string "`for UPDATE`" to place a lock on the records that are retrieved by the query.

A `TableDataSet` can be constructed with a `KeyDef`, a dbKona object used for setting a unique key for saving updates and deletes to the DBMS. If you are working with an Oracle database, you can set the `TableDataSet's KeyDef` to "`ROWID`," which is a unique key inherent in each table. Then construct the TableDataSet with a set of attributes that includes "`ROWID`."

- A `TableDataSet` is constructed in the context of a `java.sql.Connection` object; that is, you pass the `Connection` object as an argument to the `TableDataSet` constructor. Its data retrieval is specified by the name of a DBMS table. If you intend to save updates and deletes, you must supply a `KeyDef` object when the `TableDataSet` is constructed. You may refine a query with the `where()` and `order()` methods to set `WHERE` and `ORDER BY` clauses after the `TableDataSet` is created.
- A `TableDataSet` has a default `SelectStmt` object associated with it that can be used to take advantage of Query-by-example functionality.
- A `TableDataSet` is populated with `Records` (accessible by 0-origined index), which contain Values (accessible by 1-origined index).

- A `TableDataSet`'s attributes are described by a `Schema`, which stores information about the `TableDataSet`'s attributes, like column name, data type, size, and order of the database columns represented in the `TableDataSet`.
- `TableDataSets` can be cached on a WebLogic JDBC Server.
- The `setRefreshOnSave()` method sets the `TableDataSet` so that any record inserted or updated during a save is also immediately refreshed from the DBMS. Set this flag if your `TableDataSet` has columns altered by the DBMS, such as the Microsoft SQL Server IDENTITY column or a column modified by an insert or update trigger.
- The `Refresh()` methods refresh records in the `TableDataSet` that would be saved in the database, that is, records that you have changed in the `TableDataSet`. Any changes you have made to a record are lost and the record is marked clean. Records you have marked for delete are not refreshed. A record you have added to the `TableDataSet` raises an exception stating that there is no DBMS representation of the row from which to refresh.
- The `saveWithoutStatusUpdate()` methods save `TableDataSet` records to the DBMS without updating the save status of the records in the `TableDataSet`. Use these methods to save `TableDataSet` records within a transaction. If the transaction is rolled back, the records in the `TableDataSet` are consistent with the database and the transaction can be retried. After the transaction is committed, call `updateStatus()` to update the save status of records in the `TableDataSet`. Once you have saved a record with `saveWithoutStatusUpdate()`, you cannot modify it until you call `updateStatus()` on the record.
- The `TableDataSet.setOptimisticLockingCol()` method allows you to designate a single column in the `TableDataSet` as an optimistic locking column. Applications use this column to detect whether another user has changed the row since it was read from the database. dbKona assumes the DBMS updates the column whenever the row is changed, so it does not update this column from the value in the `TableDataSet`. It uses the column in the `WHERE` clause of an `UPDATE` statement when you save the record or the `TableDataSet`. If another user has modified the record, dbKona's update fails; you can retrieve the new values for the record using `Record.refresh()`, make your changes to the record, and try to save the record again.

The `TableDataSet` class (see `weblogic.db.jdbc.TableDataSet`) has methods for:

- Constructing a `TableDataSet`

- Setting its `WHERE` and `ORDER BY` clauses
- Getting its `KeyDef`
- Getting its associated JDBC `ResultSet`
- Getting its `SelectStmt`
- Getting its associated DBMS table name
- Saving its changes to a database
- Refreshing its records from the DBMS
- Getting other information about it

The superclass `DataSet` contains methods for managing record caching.

EventfulTableDataSet (Deprecated)

An `EventfulTableDataSet`, for use within WebLogic, is a `TableDataSet` that sends and receives events when its data is updated locally or in the DBMS. `EventfulTableDataSet` implements `weblogic.event.actions.ActionDef`, which is the interface implemented by all `Action` classes in WebLogic Events. The `action()` method of an `EventfulTableDataSet` updates the DBMS and notifies all other `EventfulTableDataSets` for the same DBMS table of the change. (You can read more about WebLogic Events in the White Paper and the Developer's Guide for WebLogic Events, also deprecated.)

When an `EventfulRecord` in an `EventfulTableDataSet` changes, it sends an `EventMessage` to the WebLogic Server with a `ParamSet` that contains the row that changed as well as the changed data, for the topic WEBLOGIC.[*tablename*], where the *tablename* is the name of the table associated with an `EventfulTableDataSet`. `EventfulTableDataSet` takes action on the received, evaluated event to update its own copy of the record that changed.

An `EventfulTableDataSet` is constructed in the context of a java.sql.Connection object, as an argument to the constructor. You must also supply a t3 Client object, a `KeyDef` to be used for inserts, updates, and deletes, and the name of the DBMS table.

- Like a `TableDataSet`, an `EventfulTableDataSet` has a default `SelectStmt` object associated with it that can be used to take advantage of Query-by-example functionality.

- An EventfulTableDataSet is populated with EventfulRecords (accessible by a 0-origined index). Like Records, EventfulRecords contain Values (accessible by a 1-origined index).
- An EventfulTableDataSet's attributes are described by its Schema, in the same way as a TableDataSet.

For example, an EventfulTableDataSet might be used by a warehouse inventory system to automagically update many views of a table. Here is how it works. Each warehouse employee's client application creates an EventfulTableDataSet from the "stock" table and displays those records in a Java application. Employees doing different jobs might have different displays, but all of the client applications are using an EventfulTableDataSet of the "stock" table. Because a TableDataSet is "eventful," each record in the data set has registered an interest in itself automatically. The WebLogic Topic Tree has a registration of interest for all the records; for each client, there is a registration of interest in each record in the TableDataSet.

When a user changes a record, the DBMS is updated with the new record. At the same time, an EventMessage (embedded with the changed Record itself) is automatically sent to the WebLogic Server. Each client using an EventfulTableDataSet of the "stock" table receives an event notification that has embedded in it the changed Record. The EventfulTableDataSet for each client accepts the changed Record and updates the GUI.

Record

Records are created as part of a DataSet. You can also construct Records manually in the context of a DataSet and its Schema, or the Schema of an SQL table known to an active Database session.

Records in a TableDataSet may be saved to the database individually with the save() method in the Record class, or corporately with the save() method in the TableDataSet class.

- Records are constructed when a DataSet is created and its query is executed. A Record may also be added to an existing DataSet with the DataSet.addRecord() method or with a Record constructor (after the DataSet's fetchRecords() method has been called to get its Schema).
- A Record contains a collection of Values. Records are accessible by a 0-origined index position. Values within a Record are accessible by 1-origined index position.

- A Record is described by the Schema of its parent DataSet. The Schema associated with a Record holds information about the name, data type, size, and order of each field in the Record.

The Record class (see weblogic.db.jdbc.Record) has methods for:

- Constructing a Record object
- Determining its parent DataSet and Schema
- Determining the number of columns in it
- Determining its save or update status
- Determining the SQL string used to save or update a Record to the database
- Getting and setting its Values
- Returning the value of each of its columns as a formatted string

Value

A Value object has an internal type, which is defined by the Schema of its parent DataSet. A Value object can be assigned a value with a data type other than its internal type, if the assignment is legal. A Value object can also return the value of a data type other than its internal data type, if the request is legal.

The Value object acts to shield the application from the details of manipulating vendor-specific data types. The Value object "knows" its data type, but all Value objects can be manipulated within a Java application with the same methods, no matter the internal data type.

- Values are created when Records are created.
- The internal data type of a Value object may be among the following:
  - Boolean
  - Byte
  - Byte[]
  - Date
  - Double-precision
  - Floating-point
  - Integer

- Long
- Numeric
- Short
- String
- Time
- Timestamp
- NULL

These types are mapped to the JDBC types listed in `java.sql.Types`.

- `Values` are described by the `Schema` associated with its parent `DataSet`.

The `Value` class (see `weblogic.db.jdbc.Value`) has methods for getting and setting the data and data type of a `Value` object.

Data Description Objects in dbKona

Data description objects contain metadata; that is, information about data structure, how data are stored on and retrieved from the DBMS, whether and how data can be updated. Some of the data description objects that dbKona uses are implementations of the JDBC interface; a brief description and how to use these is provided here.

- `Schema`
- `Column`
- `KeyDef`
- `SelectStmt`

Schema

When you instantiate a `DataSet`, you implicitly create the `Schema` that describes it, and when you fetch its `Records`, its `Schema` is updated.

- A `Schema` is constructed automatically when a `DataSet` is instantiated.
- A `DataSet's` attributes (and therefore, attributes of `QueryDataSets` and `TableDataSets`, and their associated Records) are defined by a `Schema`, as are the attributes of a `Table`.
- Schema attributes are described as a collection of `Column` objects.

The Schema class (see weblogic.db.jdbc.Schema) has methods for:

- Adding and returning the Columns associated with the Schema
- Determining the number of columns in a Schema
- Determining the (1-origined) index position of a particular column name in the Schema

Column

Schema is created.

The Column class (see weblogic.db.jdbc.Column) has methods for:

- Setting the Column to a particular data type
- Determining the data type of a Column
- Determining the database-specific data type of a Column
- Determining the name, scale, precision, and storage length of a Column
- Determining whether NULL values are allowed in the native DBMS column
- Determining if the Column is read-only and/or searchable

KeyDef

"WHERE attribute1 = value1 and attribute2 = value2," and so on, to uniquely identify and manipulate a particular database record. The attributes in a KeyDef should correspond to unique key in the database table.

The KeyDef object with no attributes is constructed in the KeyDef class. Use the addAttrib() method to build the attributes of the KeyDef, and then use the KeyDef as an argument in the constructor for a TableDataSet. Once the KeyDef is associated with a DataSet, you cannot add anymore attributes to it.

When you are working with an Oracle database, you can add the attribute "ROWID," which is an inherently unique key associated with each table, to be used for inserts and deletes with a TableDataSet.

The KeyDef class (see weblogic.db.jdbc.KeyDef) has methods for:

- Adding attributes
- Determining the number of attributes in it
- Determining if it has an attribute that corresponds to a particular column name or index position.

SelectStmt

A `SelectStmt` object is constructed in the `SelectStmt` class. Then add clauses to the `SelectStmt` with methods in the `SelectStmt` class, and use the resulting `SelectStmt` object as an argument when you create a `QueryDataSet`. A `TableDataSet` also has a default `SelectStmt` associated with it that can be used to further refine data retrieval after the `TableDataSet` has been created.

Methods in the `SelectStmt` class (see weblogic.db.jdbc.SelectStmt) correspond to the clauses in a SQL statement, which include:

- `Field (and an alias)`
- `From`
- `Group`
- `Having`
- `Order by`
- `Unique`
- `Where`

There is also full support for setting and adding Query-by-example clauses. Note that with the `from()` method, you can specify a string that includes an alias, in the format "`<i>tableName alias</i>`". With the `field()` method, you can use a string after the format "`<i>tableAlias.attribute</i>`" as an argument. You are not limited to a single table name when constructing a `SelectStmt` object, although its usage may dictate whether or not a join is useful. A `SelectStmt` object associated with a `QueryDataSet` can join one or more tables, whereas a `TableDataSet` cannot, since it is by definition limited to the data in a single table.

Miscellaneous Objects in dbKona

Other miscellaneous objects in dbKona include Exceptions and Constants.

- `Exceptions`

- Constants

Exceptions

- DataSetException
- LicenseException
- java.sql.SQLException

In general, DataSetExceptions occur when there is a problem with a DataSet, including errors generated from stored procedures, or when there is an internal I/O error.

java.sql.SqlExceptions are thrown when there is a problem building an SQL statement or executing it on the DBMS server.

Constants

The Enums class contains constants for the following:

- Trigger states
- Vendor-specific database types
- INSERT, UPDATE, and DELETE database operations

The java.sql.Types class contains constants for data types.

Entity Relationships

Inheritance Relationships

The following describes important descendancy relationships between dbKona classes. One class is subclassed:

DataSet
: DataSet is the abstract base class for QueryDataSet and TableDataSet.

Other dbKona objects descend from DbObject.

Most dbKona `Exceptions`, including `DataSetException` and `LicenseException`, are subclassed from `java.lang.Exception` and `weblogic.db.jdbc.DataSetException`. LicenseException is subclassed from `RuntimeException`.

Possession Relationships

Each dbKona object may have other objects associated with it that further define its structure. These relationships are described as follows.

DataSet

A `DataSet` has Records, each of which has Values. A `DataSet` has a Schema that defines its structure, which is made up of one or more Columns. A `DataSet` may have a `SelectStmt` that sets parameters for data retrieval.

TableDataSet

A `TableDataSet` has a `KeyDef` for updates and deletes by key.

Schema

A `Schema` has `Columns` that define its structure.

Implementing with dbKona

The following sections describe a set of working examples that illustrate several steps to building a simple Java application that retrieves and displays data from a remote DBMS.

Accessing a DBMS with dbKona

The following steps describe how to use dbKona to access a DBMS.

Step 1. Importing packages

Applications that use dbKona need access to `java.sql` and `weblogic.db.jdbc` (the WebLogic dbKona package), plus any other Java classes that you will use. In the following case, we also import the Properties class from `java.util`, used during the login process, and the `weblogic.html` package.

```
import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.html.*;
import java.util.Properties;
```

Note that you do *not* import the package for your JDBC driver. The JDBC driver is established during the connection phase. For version 2.0 and later, you do not import `weblogic.db.common, weblogic.db.server,` or `weblogic.db.t3client.`

Step 2. Setting Properties for Making a Connection

The following code example is a method for creating the Properties object that will be used later in this tutorial to make a connection to an Oracle DBMS. Each property is set with a double-quote-enclosed string.

```
public class tutor {

  public static void main(String argv[])
    throws DataSetException, java.sql.SQLException,
    java.io.IOException, ClassNotFoundException
   {
    Properties props = new java.util.Properties();
    props.put(“user",      “scott");
    props.put(“password",  “tiger");
    props.put(“server",    “DEMO");
    (continued below)
```

The `Properties` object will be used as an argument to create a `Connection`. The `JDBC Connection` object will become an important context for other database operations.

Step 3. Making a Connection to the DBMS

A `Connection` object is created by loading the JDBC driver class with the `Class.forName()` method, and then calling the `java.sql.myDriver.connect()` constructor, which takes two arguments, the `URL` of the JDBC driver to be used and a `java.util.Properties` object.

You can see how to create the Properties object, *props*, in step 2.

```
Driver myDriver = (Driver)
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
conn =
   myDriver.connect("jdbc:weblogic:oracle", props);
conn.setAutoCommit(false);
```

The `Connection` *conn* becomes an argument for other actions that involve the DBMS, for instance creating `DataSets` to hold query results. For details about connecting to a DBMS, see the developers guide for your your driver.

`Connections`, `DataSets` (and, if you use them, JDBC `ResultSets`), and `Statements` should be closed with the `close()` method when you have finished working with them. Note in the code examples that follow that each of these is explicitly closed.

Note: The default mode of `java.sql.Connection` sets autocommit to `true`. Oracle will perform much faster if you set `autocommit` to `false`, as shown above.

Note: `DriverManager.getConnection()` is a synchronized method, which can cause your application to hang in certain situations. For this reason, BEA recommends that you use the `Driver.connect()` method instead of `DriverManager.getConnection()`

Preparing a Query, Retrieving, and Displaying Data

The following steps describe how to prepare a query, and retrieve and display data.

Step 1. Setting Parameters for Data Retrieval

In dbKona, there are several ways to set parameters—to compose the SQL statement and set its scope—for retrieving data. Here we show how dbKona can interact at a very basic level with any JDBC driver, by taking the results of a JDBC `ResultSet` and creating a `DataSet`. In this example, we use a `Statement` object to execute a SQL statement. A `Statement` object is created with a method from the JDBC `Connection` class, and then the `ResultSet` is created by executing the Statement.

```
Statement stmt = conn.createStatement();
stmt.execute("SELECT * from empdemo");
ResultSet rs = stmt.getResultSet();
```

You can use the results of a query executed with a Statement object to instantiate a QueryDataSet. This QueryDataSet is constructed with a JDBC ResultSet:

```
Statement stmt = conn.createStatement();
stmt.execute("SELECT * from empdemo");
ResultSet rs = stmt.getResultSet();
QueryDataset ds = new QueryDataSet(rs);
```

Using the results from the execution of a JDBC Statement is only one way to create a DataSet. It requires knowledge of SQL, and it doesn't give you much control over the results of your query: basically, you can iterate through the records with the JDBC next() method. With dbKona, you do not have to know much about SQL to retrieve records; you can use methods in dbKona to set up your query, and once you have created a DataSet with your records, you have a much finer control over manipulating the records.

Step 2. Creating a DataSet for the Query Results

Instead of requiring you to compose an SQL statement, dbKona lets you use methods to set certain parts of the statement. You create a DataSet (either a TableDataSet or a QueryDataSet) for the results of the query.

For example, the simplest data retrieval in dbKona is into a TableDataSet. Creating a TableDataSet requires just a Connection object and the name of the DBMS table that you want to retrieve, as in this example that retrieves the Employee table (alias "empdemo"):

```
TableDataSet tds = new TableDataSet(conn, "empdemo");
```

A TableDataSet can be constructed with a subset of the attributes (columns) in a DBMS table. If you want to retrieve just a few columns from a very large table, specifying those columns is more efficient than retrieving the entire table. To do this, pass a list of table attributes as a string in the constructor. For example:

```
TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
dept");
```

Use a TableDataSet if you want to be able to save changes to the DBMS, or if you do not plan to do a join of one or more tables to retrieve data; otherwise, use a QueryDataSet. In this example, we use the QueryDataSet constructor that takes two arguments: a Connection object and a string that is the SQL:

```
QueryDataSet qds = new QueryDataSet(conn, "select * from
empdemo");
```

You do not actually begin receiving data until you call the `fetchRecords()` method in the `DataSet` class. After you create a `DataSet`, you can continue to refine its data parameters. For instance, we could refine the selection of records to be retrieved in the `TableDataSet` with the `where()` method, which adds a `WHERE` clause to the SQL that dbKona composes. The following retrieves just one record from the Employee table by using the `where()` method to create a `WHERE` clause:

```
TableDataSet tds = new TableDataSet(conn, "empdemo");
tds.where("empno = 8000");
```

Step 3. Fetching the Results

When you are statisfied with the data paramaters, call the `fetchRecords()` method from the `DataSet` class, as shown in this example:

```
TableDataset tds = new TableDataSet(conn, "empdemo", "empno,
dept");
tds.where("empno = 8000");
tds.fetchRecords();
```

The `fetchRecords()` method can take arguments to fetch a certain number of records, or to fetch records starting with a particular record. In the following example, we fetch no more than the first 20 records and discard the rest with the `clearRecords()` method:

```
TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
dept");
tds.where("empno > 8000");
tds.fetchRecords(20)
   .clearRecords();
```

When dealing with very large query results, you may prefer to fetch a few records at a time, process them, and then clear the `DataSet` before the next fetch. Use the `clearRecords()` method from the `DataSet` class to clear the `TableDataSet` between fetches, as illustrated here:

```
TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
dept");
tds.where("empno > 2000");
while (!tds.allRecordsRetrieved()) {
   tds.fetchRecords(100);
   // Process the hundred records . . .
   tds.clearRecords();
}
```

You can also reuse a `DataSet` with a method that was added in release 2.5.3. This method, `DataSet.releaseRecords()`, closes the `DataSet` and releases all the Records but does not nullify them. You can reuse the `DataSet` to generate new records, yet any records from the first use still held by the application remain readable.

Step 4. Examining a TableDataSet's Schema

Here is a simple example of how you can examine the `Schema` information for a `TableDataSet`. The `toString()` method in the `Schema` class displays a newline-delimited list of the name, type, length, precision, scale, and null-allowable attributes of the columns in the table queried for a `TableDataSet` *tds*:

```
Schema sch = tds.schema();
System.out.println(sch.toString());
```

If you use a Statement object to create a query, you should close the Statement after you have completed the query and fetched its results:

```
stmt.close();
```

Step 5. Examining the Data with htmlKona

The following example shows how you might use an htmlKona `UnorderedList` to examine the data. This example uses `DataSet.getRecord()` and `Record.getValue()` to examine each record in a for loop. This finds the name, ID, and salary of the employee making the most money from the records retrieved in the QueryDataSet we created in step 2:

```
// (Creation of Database session object and QueryDataSet qds)
UnorderedList ul = new UnorderedList();

String name     = "";
String id       = "";
String salstr   = "";
int sal         = 0;
for (int i = 0; i < qds.size(); i++) {
  // Get a record
  Record rec = qds.getRecord(i);
  int tmp = rec.getValue("Emp Salary").asInt();
  // Add the salary amount to the htmlKona ListElement
  ul.addElement(new ListItem("$" + tmp));
 // Compare this salary to the maximum salary we have found so far
  if (tmp > sal) {
    // If this salary is a new max, save away the employee's info
```

```
    sal    = tmp;
    name   = rec.getValue("Emp Name").asString();
    id     = rec.getValue("Emp ID").asString();
    salstr = rec.getValue("Emp Salary").asString();
  }
```

Step 6. Displaying the Results with htmlKona

htmlKona provides a convenient way to display dynamic data like that produced by the above example. The following example shows how you might construct a page on the fly for displaying the results of your query:

```
HtmlPage hp = new HtmlPage();
hp.getHead()
  .addElement(new TitleElement("Highest Paid Employee"));
hp.getBodyElement()
  .setAttribute(BodyElement.bgColor, HtmlColor.white);
hp.getBody()
  .addElement(MarkupElement.HorizontalLine)
  .addElement(new HeadingElement("Query String: ", +2))
  .addElement(stmt.toString())
  .addElement(MarkupElement.HorizontalLine)
  .addElement("I examined the values: ")
  .addElement(ul)
  .addElement(MarkupElement.HorizontalLine)
  .addElement("Max salary of those employees examined is: ")
  .addElement(MarkupElement.Break)
  .addElement("Name: ")
  .addElement(new BoldElement(name))
  .addElement(MarkupElement.Break)
  .addElement("ID: ")
  .addElement(new BoldElement(id))
  .addElement(MarkupElement.Break)
  .addElement("Salary: ")
  .addElement(new BoldElement(salstr))
  .addElement(MarkupElement.HorizontalLine);

hp.output();
```

Step 7. Closing the DataSet and the Connection

```
qds.close();
tds.close();
```

It is also important to close the Connection to the DBMS. This code should appear at the end of all of your database operations in a finally block, as in this example:

```
  try {
  // Do your work
  }
  catch (Exception mye) {
  // Catch and handle exceptions
  }
  finally {
    try {conn.close();}
    catch (Exception e) {
      // Deal with any exceptions
    }
  }
```

Code summary

```
import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.html.*;
import java.util.Properties;

public class tutor {

  public static void main(String[] argv)
       throws java.io.IOException, DataSetException,
       java.sql.SQLException, HtmlException,
       ClassNotFoundException
  {
  Connection conn = null;
  try {
    Properties props = new java.util.Properties();
    props.put("user",      "scott");
    props.put("password",  "tiger");
    props.put("server",    "DEMO");

    Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();
    conn =
      myDriver.connect("jdbc:weblogic:oracle",
                                    props);
    conn.setAutoCommit(false);

    // Create a TableDataSet to add 10 records
    TableDataSet tds = new TableDataSet(conn, "empdemo");
    for (int i = 0; i < 10; i++) {
      Record rec = tds.addRecord();
      rec.setValue("empno", i)
         .setValue("ename", "person " + i)
         .setValue("esalary", 2000 + (i * 10));
```

```
}

// Save the data and close the TableDataSet
tds.save();
tds.close();

// Create a QueryDataSet to retrieve the additions to the table
Statement stmt = conn.createStatement();
stmt.execute("SELECT * from empdemo");

QueryDataSet qds = new QueryDataSet(stmt.getResultSet());
qds.fetchRecords();

// Use the data from the QueryDataSet
UnorderedList ul = new UnorderedList();

String name     = "";
String id       = "";
String salstr   = "";
int sal         = 0;
for (int i = 0; i < qds.size(); i++) {
  Record rec = qds.getRecord(i);
  int tmp = rec.getValue("Emp Salary").asInt();
  ul.addElement(new ListItem("$" + tmp));
  if (tmp > sal) {
    sal    = tmp;
    name   = rec.getValue("Emp Name").asString();
    id     = rec.getValue("Emp ID").asString();
    salstr = rec.getValue("Emp Salary").asString();
  }
}

// Use an htmlKona page to display the data retrieved, and the
// statements used to retrieve it
HtmlPage hp = new HtmlPage();
hp.getHead()
  .addElement(new TitleElement("Highest Paid Employee"));
hp.getBodyElement()
  .setAttribute(BodyElement.bgColor, HtmlColor.white);
hp.getBody()
  .addElement(MarkupElement.HorizontalLine)
  .addElement(new HeadingElement("Query String: ", +2))
  .addElement(stmt.toString())
  .addElement(MarkupElement.HorizontalLine)
  .addElement("I examined the values: ")
  .addElement(ul)
  .addElement(MarkupElement.HorizontalLine)
  .addElement("Max salary of those employees examined is: ")
  .addElement(MarkupElement.Break)
```

```
        .addElement("Name: ")
        .addElement(new BoldElement(name))
        .addElement(MarkupElement.Break)
        .addElement("ID: ")
        .addElement(new BoldElement(id))
        .addElement(MarkupElement.Break)
        .addElement("Salary: ")
        .addElement(new BoldElement(salstr))
        .addElement(MarkupElement.HorizontalLine);

      hp.output();

      // Close QueryDataSet
      qds.close();
      }
      catch (Exception e) {
        // Deal with any exceptions
      }
      finally {
      // Close the connection
        try {conn.close();}
        catch (Exception mye) {
          // Deal with any exceptions
        }
      }
    }
  }
```

Note that we closed each `Statement` and `DataSet` after use, and that we closed the `Connection` in a `finally` block.

Using a SelectStmt Object to Form a Query

The following steps describe how to form a query using a `SelectStmt` object.

Step 1. Setting SelectStmt Parameters

When you create a `TableDataSet`, it is associated with an empty `SelectStmt` that you can then modify to form a query. In this example, we have already created a connection *conn*. Here is how you access a `TableDataSet's SelectStmt`:

```
TableDataSet tds = new TableDataSet(conn, "empdemo");
SelectStmt sql = tds.selectStmt();
```

Now set the parameters for the `SelectStmt` object. In the example, the first argument for each field is the attribute name and the second is the alias. This query will retrieve information about all employees who make less than $2000:

```
sql.field("empno", "Emp ID")
   .field("ename", "Emp Name")
   .field("sal", "Emp Salary")
   .from("empdemo")
   .where("sal < 2000")
   .order("empno");
```

Step 2. Using QBE to Refine the Parameters

The `SelectStmt` object also gives you `Query-by-example` functionality. `Query-by-example`, or `QBE`, forms parameters for data retrieval using a set of phrases that follow the format column operator value. For example, "empno = 8000" is a `Query-by-example` phrase that can select all the rows in one or more tables where the field employee number (`"empno", alias "Emp ID"`) equals 8000.

We can further define the parameters for data selection by using the `setQbe()` and `addQbe()` methods in the `SelectStmt` class, as is shown here. These methods allow you to use vendor-specific `QBE` syntax in constructing a select statement:

```
sql.setQbe("ename", "MURPHY")
   .addUnquotedQbe("empno", "8000");
```

When you have finished, use the `fetchRecords()` method to populate the `DataSet`, as we did in the second tutorial.

Modifying DBMS Data with a SQL Statement

The following steps describe how to modify DBMS data with a SQL statement.

Step 1. Writing SQL Statements

When you retrieve data that you expect to modify, and if you want to save those modifications into the remote DBMS, you should retrieve data into a `TableDataSet`. Changes made to `QueryDataSets` cannot be saved.

As with most dbKona operations, you should begin by creating the `Properties` and `Driver` objects, and then instantiating a `Connection`.

Step 1. Writing SQL statements

```
String insert = "insert into empdemo(empno, " +
                "ename, job, deptno) values " +
                "(8000, 'MURPHY', 'SALESMAN', 10)";
```

The second statement changes Murphy's name to Smith, and changes his job status from Salesman to Manager:

```
String update = "update empdemo set ename = 'SMITH', " +
                "job = 'MANAGER' " +
                "where empno = 8000";
```

The third statement deletes this record from the database:

```
String delete = "delete from empdemo where empno = 8000";
```

Step 2. Executing Each SQL Statement

First, save a snapshot of the table into a `TableDataSet`. Later we'll examine each `TableDataSet` to verify that the execute operation produced the expected results. Notice that `TableDataSets` are instantiated with the results of an executed query.

```
Statement stmt1 = conn.createStatement();
stmt1.execute(insert);

TableDataSet ds1 = new TableDataSet(conn, "emp");
ds1.where("empno = 8000");
ds1.fetchRecords();
```

The methods associated with `TableDataSet` allow you to specify a SQL `WHERE` clause and a SQL `ORDER BY` clause and to set and add to a `QBE` statement. We use the `TableDataSet` in this example to requery the database table "`emp`" after each statement is executed to see the results of the `execute()` method. With the "`where`" clause, we narrow down the records in the table to just employee number 8000.

Repeat the `execute()` method for the update and delete statements and capture the results into two more `TableDataSets`, *ds2* and *ds3*.

Step 3. Displaying the Results with htmlKona

```
ServletPage hp = new ServletPage();
hp.getHead()
  .addElement(new TitleElement("Modifying data with SQL"));
hp.getBody()
```

```
    .addElement(MarkupElement.HorizontalLine)
    .addElement(new TableElement(tds))
    .addElement(MarkupElement.HorizontalLine)
   .addElement(new HeadingElement("Query results afer INSERT", 2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(insert))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds1))
    .addElement(MarkupElement.HorizontalLine)
   .addElement(new HeadingElement("Query results after UPDATE", 2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(update))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds2))
    .addElement(MarkupElement.HorizontalLine)
   .addElement(new HeadingElement("Query results after DELETE", 2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(delete))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds3))
    .addElement(MarkupElement.HorizontalLine);
  hp.output();
```

Code summary

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;
import weblogic.db.jdbc.*;
import weblogic.html.*;

public class InsertUpdateDelete extends HttpServlet {

  public synchronized void service(HttpServletRequest req,
                                   HttpServletResponse res)
    throws IOException
  {
    Connection conn = null;
    try {
      res.setStatus(HttpServletResponse.SC_OK);
      res.setContentType("text/html");

      Properties props = new java.util.Properties();
      props.put("user",      "scott");
      props.put("password",  "tiger");
      props.put("server",    "DEMO");
```

```
Driver myDriver = (Driver)
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
conn =
  myDriver.connect("jdbc:weblogic:oracle",
                                 props);
conn.setAutoCommit(false);

// Create a TableDataSet with a SelectStmt
TableDataSet tds = new TableDataSet(conn, "empdemo");
SelectStmt sql = tds.selectStmt();
sql.field("empno", "Emp ID")
   .field("ename", "Emp Name")
   .field("sal", "Emp Salary")
   .from("empdemo")
   .where("sal < 2000")
   .order("empno");
sql.setQbe("ename", "MURPHY")
   .addUnquotedQbe("empno", "8000");
tds.fetchRecords();

String insert = "insert into empdemo(empno, " +
                "ename, job, deptno) values " +
                "(8000, 'MURPHY', 'SALESMAN', 10)";

// Create a statement and execute it
Statement stmt1 = conn.createStatement();
stmt1.execute(insert);
stmt1.close();

// Verify results
TableDataSet ds1 = new TableDataSet(conn, "empdemo");
ds1.where("empno = 8000");
ds1.fetchRecords();

// Create a statement and execute it
String update = "update empdemo set ename = 'SMITH', " +
                "job = 'MANAGER' " +
                "where empno = 8000";
Statement stmt2 = conn.createStatement();
stmt2.execute(insert);
stmt2.close();

// Verify results
TableDataSet ds2 = new TableDataSet(conn, "empdemo");
ds2.where("empno = 8000");
ds2.fetchRecords();

// Create a statement and execute it
String delete = "delete from empdemo where empno = 8000";
```

```
        Statement stmt3 = conn.createStatement();
        stmt3.execute(insert);
        stmt3.close();

        // Verify results
        TableDataSet ds3 = new TableDataSet(conn, "empdemo");
        ds3.where("empno = 8000");
        ds3.fetchRecords();

        // Create a servlet page to display the results
        ServletPage hp = new ServletPage();
        hp.getHead()
          .addElement(new TitleElement("Modifying data with SQL"));
        hp.getBody()
          .addElement(MarkupElement.HorizontalRule)
          .addElement(new HeadingElement("Original table", 2))
          .addElement(new TableElement(tds))
          .addElement(MarkupElement.HorizontalRule)
         .addElement(new HeadingElement("Query results afer INSERT",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(insert))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds1))
          .addElement(MarkupElement.HorizontalRule)
        .addElement(new HeadingElement("Query results after UPDATE",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(update))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds2))
          .addElement(MarkupElement.HorizontalRule)
        .addElement(new HeadingElement("Query results after DELETE",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(delete))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds3))
          .addElement(MarkupElement.HorizontalRule);

        hp.output();

        tds.close();
        ds1.close();
        ds2.close();
        ds3.close();
      }
      catch (Exception e) {
        // Handle the exception
```

```
    }
    // Always close the connection in a finally block
    finally {
      conn.close();
    }
  }
}
```

Modifying DBMS Data with a KeyDef

Use a `KeyDef` object to establish keys for deleting and inserting data into the remote DBMS. A `KeyDef` acts as an equality statement in updates and deletes after the pattern `WHERE KeyDef attribute1 = value1` and `KeyDef attribute2 = value2`, and so on.

The first step is to create a connection to the DBMS. In this example, we use the `Connection` object *conn* created in the first tutorial. The database table we use in this example is the `Employee` table ("`empdemo`"), with fields `empno`, `ename`, `job`, and `deptno`. The query we execute retrieves the full contents of the table `empdemo`.

Step 1. Creating a KeyDef and Building Its Attributes

The KeyDef object we create for inserts and deletes in this tutorial has one attribute, the `empno` column in the database. Creating a `KeyDef` with this attribute will set a key after the pattern `WHERE empno =` and the particular value assigned to `empno` for each record to be saved.

A `KeyDef` is created and built in the `KeyDef` class, as shown in this example:

```
KeyDef key = new KeyDef().addAttrib("empno");
```

If you are working with an Oracle database, you can construct the `KeyDef` with the attribute "`ROWID`," to do inserts and deletes on this Oracle key, as in this example:

```
KeyDef key = new KeyDef().addAttrib("ROWID");
```

Step 2. Creating a TableDataSet with a KeyDef

In this example, we create a `TableDataSet` with the results of our query. We use the `TableDataSet` constructor that takes a Connection object, a DBMS table name, and a `KeyDef` as its arguments:

```
TableDataSet tds = new TableDataSet(conn, "empdemo", key);
```

The KeyDef becomes the reference for all changes that we will make to the data. Each time we save the TableDataSet, we change data in the database (according to the limits set on SQL UPDATE, INSERT, and DELETE operations) based on the value of the KeyDef attribute, which in this example is the employee number ("empno").

If you are working with an Oracle database and have added the attribute ROWID to the KeyDef, you can construct a TableDataSet for inserts and deletes like this:

```
KeyDef key = new KeyDef().addAttrib("ROWID");
TableDataSet tds =
   new TableDataSet(conn, "empdemo", "ROWID, dept", key);
tds.where("empno < 100");
tds.fetchRecords();
```

Step 3. Inserting a Record into the TableDataSet

You can create a new Record object in the context of the TableDataSet to which it is to be added with the addRecord() method from the TableDataSet class. Once you have added the record, you can set the values for each of its fields with the setValue() method from the Record class. You must set at least one value in a new Record if you intend to save it into the database: the KeyDef field:

```
Record newrec = tds.addRecord();
newrec.setValue("empno",  8000)
      .setValue("ename",  "MURPHY")
      .setValue("job",    "SALESMAN")
      .setValue("deptno", 10);
String insert = newrec.getSaveString();
tds.save();
```

The getSaveString() method in the Record class returns the SQL string (a SQL UPDATE, DELETE, or INSERT statement) used to save a Record to the database. We saved this string into an object that we can display later to examine exactly how the insert operation was carried out.

Step 4. Updating a Record in the TableDataSet

You also use the setValue() method to update a Record. In the following example, we'll make a change to the record we created in the previous step:

```
newrec.setValue("ename", "SMITH")
      .setValue("job",   "MANAGER");
```

```
String update = newrec.getSaveString();
tds.save();
```

Step 5. Deleting a Record from the TableDataSet

You can mark a record in a TableDataSet for deletion with the `markToBeDeleted()` method (or unmark it with the `unmarkToBeDeleted()` method) in the `Record` class. For instance, deleting the record we just created would be accomplished by marking the record for deletion, as shown here:

```
newrec.markToBeDeleted();
String delete = newrec.getSaveString();
tds.save();
```

Records marked for deletion are not removed from a `TableDataSet` until you `save()` it, or until you execute the `removeDeletedRecords()` method in the `TableDataSet` class.

Records that have been removed from the `TableDataSet` but not yet deleted from the database (by the `removeDeletedRecords()` method) fall into a zombie state. You can determine whether a record is a zombie by testing it with the `isAZombie()` method in the Record class, as shown here:

```
if (!newrec.isAZombie()) {
 . . .
}
```

Step 6. More on Saving the TableDataSet

Saving a `Record` or a `TableDataset` will effectively save the data to the database. dbKona performs selective changes, that is, only data that has changed is saved. Inserting, updating, and deleting records in the `TableDataSet` affects only the data in the `TableDataSet` until you use the `Record.save()` or `TableDataSet.save()` method.

Checking Record Status Before Saving

Several methods from the `Record` class return information about the state of a `Record` that you may want to know before a `save()` operation. Some of these are:

`needsToBeSaved()` and `recordIsClean()`
: Use the `needsToBeSaved()` method to determine whether a `Record` needs to be saved, that is, whether it has been changed since it was retrieved or last saved. The `recordIsClean()` method determines whether any of the `Values` in a `Record` need to be saved. This method just determines whether a `Record` is dirty, no matter whether the scheduled database action is insert, update, or delete. Regardless of the type (insert/update/delete), the `needsToBeSaved()` method will return `false` after a `save()` operation.

`valueIsClean(int)`
: Determines whether the `Value` at a particular index position in the `Record` needs to be saved. This method takes the index position of a `Value` as its argument.

`toBeSavedWith...()`
: You can check *how a Record will be saved* with a particular SQL action with the methods `toBeSavedWithDelete()`, `toBeSavedWithInsert()`, and `toBeSavedWithUpdate()` methods. The semantics of these methods equate to the answer to the question, "If this row is or becomes dirty, what action will be taken when the `TableDataSet` is saved?"

If you want to know whether a row will participate in a save to the DBMS, use the `isClean()` and the `needsToBeSaved()` methods.

When you make modifications to a `Record` or `TableDataSet`, use the `save()` method from either class to save the changes to the database. In the previous steps, we saved the `TableDataSet` after each transaction as shown below:

```
tds.save();
```

Step 7. Verifying the changes

Here is the sample code for fetching just a single record, which is an efficient way to verify single-record changes. In this example, we use a `TableDataSet` with a `query-by-example` (`QBE`) clause to fetch just the record we're interested in:

```
TableDataSet tds2 = new TableDataSet(conn, "empdemo");
tds2.where("empno = 8000")
    .fetchRecords();
```

As a final step, we can display the query results after each step and the strings "`insert`", "`update`", and "`delete`" that we created after each `save()`. Refer to the code summary in the previous tutorial to use htmlKona for displaying the results.

When you have finished with the `DataSets`, close each one with the `close()` method:

```
tds.close();
tds2.close();
```

Code Summary

Here is a code example that uses some of the concepts covered in this section:

```
package tutorial.dbkona;

import weblogic.db.jdbc.*;
import java.sql.*;
import java.util.Properties;

public class rowid {

  public static void main(String[] argv)
    throws Exception
  {
    Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();
    conn =
      myDriver.connect("jdbc:weblogic:oracle:DEMO",
                                "scott",
                                "tiger");

    // Here we insert 100 records.
    TableDataSet ts1 = new TableDataSet(conn, "empdemo");
    for (int i = 1; i <= 100; i++) {
      Record rec = ts1.addRecord();
      rec.setValue("empid", i)
         .setValue("name",  "Person " + i)
         .setValue("dept",  i);
    }

    // Save new records. dbKona does selective saves, that is,
    // it saves only those records in the TableDataSet that have
    // changed to cut down on network traffic and server calls.
    System.out.println("Inserting " + ts1.size() + " records.");
    ts1.save();
    // Close the DataSet now that we're finished with it.
    ts1.close();

    // Define a KeyDef for updates and deletes.
    // ROWID is an Oracle specific field which can act as a
    // primary key for updates and deletes
    KeyDef key = new KeyDef().addAttrib("ROWID");
```

```
    // Update the 100 records we originally added.
    TableDataSet ts2 =
      new TableDataSet(conn, "empdemo", "ROWID, dept", key);
    ts2.where("empid <= 100");
    ts2.fetchRecords();

    for (int i = 1; i <= ts2.size(); i++) {
      Record rec = ts2.getRecord(i);
      rec.setValue("dept", i + rec.getValue("dept").asInt());
    }

    // Save the updated records.
    System.out.println("Update " + ts2.size() + " records.");
    ts2.save();

    // Delete the same 100 records.
    ts2.reset();
    ts2.fetchRecords();

    for (int i = 0; i < ts2.size(); i++) {
      Record rec = ts2.getRecord(i);
      rec.markToBeDeleted();
    }

    // Delete records from server.
    System.out.println("Delete " + ts2.size() + " records.");
    ts2.save();

    // You should always close DataSets, ResultSets, and
    // Statements when you have finished working with them.
    ts2.close();

    // Finally, make sure you close the connection.
    conn.close();
  }
}
```

Using a JDBC PreparedStatement with dbKona

Part of the convenience of dbKona is that you do not need to know much about how to write vendor-specific SQL, since dbKona will compose syntactically correct SQL for you. In some cases, however, you may want to use a JDBC `PreparedStatement` object with dbKona.

A JDBC PreparedStatement is used to precompile a SQL statement that will be used multiple times. You can clear the parameters for a PreparedStatement with a call to PreparedStatement.clearParameters().

A PreparedStatment object is constructed with the preparedStatement() method in the JDBC Connection class (the object used as *conn* in all of these examples). In this example, we create a PreparedStatement and then execute it within a loop. This statement has three IN parameters, employee id, name, and department. This will add 100 employees to the table:

```
String inssql = "insert into empdemo(empid, " +
                "name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);

for (int i = 1; i <= 100; i++) {
   pstmt.setInt(1, i);
   pstmt.setString(2, "Person" + i);
   pstmt.setInt(3, i);
   pstmt.executeUpdate();
}

pstmt.close();
```

You should always close a Statement or PreparedStatement object when you have finished working with it.

You can accomplish the same task with dbKona without worrying about the SQL. Use a KeyDef to set fields for update or delete. Check the tutorial “Modifying DBMS Data with a KeyDef” on page 7-31 for details.

Using Stored Procedures with dbKona

Access to the functionality of procedures and functions stored on a remote machine that can carry out specific, often system-independent or vendor-independent tasks extends the power of dbKona. Using stored procedures and functions requires an understanding of how requests are passed back and forth between the dbKona Java application and the remote machine. Executing a stored procedure or function changes the value of a supplied parameter. The execution of a stored procedure or function also returns a value that indicates its success or failure.

The first step, as in any dbKona application, is to connect to the DBMS. The example code uses the same Connection object, conn, that we created in the first tutorial topic.

Step 1. Creating a Stored Procedure

We use a JDBC `Statement` object to create a stored procedure by executing a call to `CREATE` on the DBMS. In this example, parameter "`field1`" is declared as an input and output parameter of type `integer`:

```
Statement stmtl = conn.createStatement();
stmtl.execute("CREATE OR REPLACE PROCEDURE proc_squareInt " +
              "(field1 IN OUT INTEGER, " +
              " field2 OUT INTEGER) IS " +
              "BEGIN field1 := field1 * field1; " +
              "field2 := field1 * 3; " +
              "END proc_squareInt;");
stmtl.close();
```

Step 2. Setting parameters

`prepareCall()` method in the JDBC `Connection` class.

In this example, we use the `setInt()` method to set the first parameter to the integer "3". Then we register the second parameter as an `OUT` parameter of type `java.sql.Types.INTEGER`. Finally, we execute the stored procedure:

```
CallableStatement cstmt =
  conn.prepareCall("BEGIN proc_squareInt(?, ?): END;");
cstmt.setInt(1, 3);
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.execute();
```

Note that Oracle does not natively support binding to "?" values in a SQL statement. Instead it uses ":1", ":2", etc. We allow you to use either in your SQL.

Step 3. Examining the Results

Let's use the simplest method and print the results to the screen:

```
System.out.println(cstmt.getInt(1));
System.out.println(cstmt.getInt(2));
cstmt.close();
```

Using Byte Arrays for Images and Audio

You can store and retrieve binary large object files from a database with a byte array. Being able to handle large database data like image and sound files is necessary for multimedia applications, which often manage data in a database.

You will probably also find htmlKona useful, which will make it easy to integrate database data retrieved with dbKona into an HTML environment. The example code that we use in this tutorial depends on htmlKona.

Step 1. Retrieving and Displaying Image Data

In this example, we use server-side Java running on a Netscape server posted from an htmlKona form to retrieve the name of the image that the user wants to view. With that image name, we query the contents of a database table called "`imagetable`" and get the first record of the results. You will notice that we use a `SelectStmt` object to construct a SQL query by `QBE`.

After we retrieve the image record, we set the HTML page type to the image type and then retrieve the image data as an array of bytes (`byte[]`) into an htmlKona `ImagePage`, which will display the image in a browser:

```
if (iname != null) {
  // Retrieve the image from the database
  TableDataSet tds = new TableDataSet(conn, "imagetable");
  tds.selectStmt().setQbe("name", iname);
  tds.fetchRecords();

  Record rec = tds.getRecord(0);

  this.returnNormalResponse("image/" +
                            rec.getValue("type").asString());

  ImagePage hp = new ImagePage(rec.getValue("data").asBytes());
  hp.output(getOutputStream());
}
```

Step 2. Inserting an Image into a Database

We can also use dbKona to insert image files into a database. Here is a snippet of code that adds two images as type array objects to a database by adding a `Record` for each image to a `TableDataSet`, setting the `Values` of the `Record`, and then saving the `TableDataSet`:

```
TableDataSet tds = new TableDataSet(conn, "imagetable");
Record rec = tds.addRecord();
rec.setValue("name", "vars")
   .setValue("type", "gif")
   .setValue("data", "c:/html/api/images/variables.gif");

rec = tds.addRecord();
rec.setValue("name", "excepts")
   .setValue("type", "jpeg")
   .setValue("data", "c:/html/api/images/exception-index.jpg");

tds.save();
tds.close();
```

Using dbKona for Oracle Sequences

dbKona provides a wrapper—a `Sequence` object—to access the functionality of Oracle sequences. An Oracle sequence is created in dbKona by supplying the starting number and increment interval for the sequence.

The following sections describe how to use dbKona for Oracle sequences.

Constructing a dbKona Sequence Object

You construct a `Sequence` object with a JDBC Connection and the name of a sequence that already exists on an Oracle server. Here is an example:

```
Sequence seq = new Sequence(conn, "mysequence");
```

Creating and Destroying Sequences on an Oracle Server from dbKona

If the Oracle sequence does not exist, you can create it from dbKona with the `Sequence.create()` method, which takes four arguments: a JDBC `Connection`, a name for the sequence to be created, an increment interval, and a starting point. Here is an example that creates an Oracle sequence "mysequence" beginning at 1000 and increasing in increments of 1:

```
Sequence.create(conn, "mysequence", 1, 1000);
```

You can drop an Oracle sequence from dbKona, also, as in this example:

```
Sequence.drop(conn, "mysequence");
```

Using a Sequence

Once you have created a `Sequence` object, you can use it to generate autoincrementing `int`s, for example, to set an autoincrementing key as you add records to a table. Use the `nextValue()` method to return an `int` that is the next increment in the `Sequence`. For example:

```
TableDataSet tds = new TableDataSet(conn, "empdemo");
  for (int i = 1; i <= 10; i++) {
      Record rec = tds.addRecord();
      rec.setValue("empno", seq.nextValue());
  }
```

You can check the current value of a Sequence with the `currentValue()` method, but only after you have called the `nextValue()` method at least once:

```
System.out.println("Records 1000-" + seq.currentValue() + "
added.");
```

Code Summary

Here is a working code example that illustrates how to use concepts discussed in this section. First, we attempt to drop a sequence named "`testseq`" from the Oracle server; this insures that we do not get an error when we try to create a sequence if one already exists by that name. Then we create a sequence on the server, and use its name to create a dbKona `Sequence` object:

```
package tutorial.dbkona;

import weblogic.db.jdbc.*;
```

```
import weblogic.db.jdbc.oracle.*;
import java.sql.*;
import java.util.Properties;

public class sequences {

  public static void main(String[] argv)
    throws Exception
  {
    Connection conn = null;
    Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();
    conn =
      myDriver.connect("jdbc:weblogic:oracle:DEMO",
                                  "scott",
                                  "tiger");

    // Drop the sequence if it already exists on the server.
    try {Sequence.drop(conn, "testseq");} catch (Exception e) {;}

    // Create a new sequence on the server.
    Sequence.create(conn, "testseq", 1, 1);

    Sequence seq = new Sequence(conn, "testseq");

    // Print out the next value in the sequence in a loop.
    for (int i = 1; i <= 10; i++) {
      System.out.println(seq.nextValue());
    }

    System.out.println(seq.currentValue());

    // Drop the sequence from the server
    // and close the Sequence object.
    Sequence.drop(conn, "testseq");
    seq.close();

    // Finally, close the connection.
    conn.close();
  }
}
```

8 Testing JDBC Connections and Troubleshooting

The following sections describe how to test JDBC connections and provide troubleshooting tips:

Testing Connections

The following sections describe how to test connections.

Validating a DBMS Connection from the Command Line

BEA provides utilities that you can use to test two-tier and three-tier JDBC database connections after you install WebLogic two-tier drivers, WebLogic Server, or WebLogic JDBC.

How to Test a Two-Tier Connection from the Command Line

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

- For Type 2 JDBC drivers, such as WebLogic jDriver for Oracle, set your `PATH` (Windows NT) 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`.
- Installation instructions for the BEA WebLogic jDriver JDBC drivers are available at:
  - Installing WebLogic jDriver for Oracle
  - Installing WebLogic jDriver for Microsoft SQL Server
  - Installing WebLogic jDriver for Informix

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, MSSQLSERVER4, or INFORMIX4`

user

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

password

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

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`, `sqlplus`, or `DBACCESS`. Type 4 drivers, such as MSSQLServer4 and Informix4, 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.

Informix (Type 4 driver)

Connect to Informix from Java with WebLogic jDriver for Informix using the same values that you use with DBACCESS. The order of arguments follows the pattern:

```
$ java utils.dbping INFORMIX user pass db@server:port
```

As shown in this example:

```
$ java utils.dbping INFORMIX bill secret stores@myserver:8543
```

How to Validate a Multitier WebLogic JDBC Connection from the Command Line

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

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

Syntax

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

Arguments

URL

URL of the WebLogic Server.

username

Valid username for the DBMS.

password

Valid password for that user.

DB

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

driver_class

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

driver_URL

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

Examples

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

Oracle

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

```
$ java utils.t3dbping          // command
    t3://bigbox:7001           // WebLogic URL
    scott tiger                // user password
```

```
        DEMO20                      // DB
        weblogic.jdbc.oci.Driver    // driver class
        jdbc:weblogic:oracle        // driver URL
```

DB2 with AS/400 Type 4 JDBC driver

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

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

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

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

```
$ java utils.t3dbping                       // command
     t3://localhost:7001                    // WebLogic URL
     sa                                     // user name
     abcd                                   // password
     hostname                       // database@hostname:port
                                    //(optional if specified
                                    // as part of the URL)
weblogic.jdbc.mssqlserver4.Driver // driver class
jdbc:weblogic:mssqlserver4:pubs@localhost:1433
                // driver URL:database@hostname:port
                //(optional if used in the database parameter)
```

Troubleshooting JDBC

The following sections provide troubleshooting tips.

Troubleshooting JDBC Connections

If you are testing a connection to WebLogic, check the WebLogic log. By default, the log is kept in a file called `weblogic.log` in the `weblogic/myserver` directory.

UNIX Users

If you encounter a problem trying to load native_login, use `truss` to determine the source of the problem. For example, to run tutorial.example3, enter:

```
$ truss -f -t open -s\!all java tutorial.example3
```

WinNT

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`.

SEGVs with JDBC and Oracle Databases

Several conditions can cause segmentation violation errors (SEGVs) or hangs when you use JDBC and an Oracle database.

- You must upgrade to the current client libraries, as specified in BEA WebLogic Server Platform Support at http://e-docs.bea.com/wls/certifications/certs_610/index.html.
- You may be using WebLogic classes with a mismatched version of the `.dll`, `.sl`, or `.so` for WebLogic jDriver for Oracle. *You must always use the* `.dll`,

`.so`, or `.sl` *file that was shipped with a particular version of the WebLogic distribution.*

- You may have exhausted the available connections in a connection pool. Make sure that your program calls the `close()` method on the connection after you are finished with it. If you need more connections, increase the size of the pool.
- If the Oracle server and WebLogic are running on the same host, and you are using an IPC connection to Oracle, the version of your client libraries *must* match the version of your server. Note that when server and client are on the same host, sqlnet will by default, attempt to make an IPC connection. You can prevent this by specifying `"automatic_ipc"=off` in your `sqlnet.ora` file.
- Your `ORACLE_HOME` environment variable may not be set correctly. You must set `ORACLE_HOME` correctly so that the OCI libraries can locate needed resource files.

Out-of-Memory Errors

A common cause of out-of-memory errors is failing to close ResultSets. The error message is usually similar to the following:

```
Run-time exception error; current exception: xalloc

No handler for exception
```

When using array fetches, the native layer allocates memory in C, not in Java, so Java garbage collection does not immediately clean up the memory. The only way to release the memory is to close the ResultSet. (You can minimize this memory usage for better performance.)

To avoid out-of-memory errors, *make sure that your program logic closes all ResultSets in all cases.* To test whether failing to close ResultSets is causing the out-of-memory errors, minimize the size of the array fetches so that the amount of C memory allocated for selects is small. You can do this by setting the `weblogic.oci.cacheRows` property (a JDBC connection property) to a small number. For example,

```
Properties props = new java.util.Properties();
props.put("user",                  "scott");
props.put("password",              "tiger");
```

```
   props.put("server",                 "DEMO" );
   props.put("weblogic.oci.cacheRows", "1"    );

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

Connection conn = d.connect("jdbc:weblogic:oracle", props);
```

If the out of memory errors cease, it is likely that `ResultSets` are not being closed somewhere in your code. For more information, see Closing JDBC Objects.

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.

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

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

```
try {

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

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

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

    }

    catch (Exception e) {

      // deal with any exceptions appropriate
```

```
}

finally {

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

}
```

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.

Behavior for the second line varies depending on which service pack of WebLogic Server you are running. Before WebLogic Server 6.1SP5, the server would return a clone of the original object, which was still subject to garbage collection. After 6.1SP5, WebLogic Server returns the original object and does not garbage collect the object until it is no longer used.

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 software) on your system. This document describes problems you may encounter and suggests solutions for them.

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

WebLogic jDriver for Oracle

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

Solaris

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

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

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

HP-UX

Incorrectly Set File Permissions

The shared library problem you are most likely to encounter after installing WebLogic on an HP-UX system is incorrectly set file permissions. After installing WebLogic, 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 weblogic_directory/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 `weblogic_directory/lib/hpux11` is in the `SHLIB_PATH` environment variable:

```
% echo $SHLIB_PATH
```

If the directory is not listed, add it:

```
# setenv SHLIB_PATH weblogic_directory/lib/hpux11:$SHLIB_PATH
```

Alternatively, copy (or link) the .sl files from the WebLogic 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 libweblogicoci37.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=
$WL_HOME/lib/hpux11/oci901_8:ORACLE/lib32:$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/docs61/oracle/install_jdbc.html`.