



BEA WebLogic Server™ and WebLogic Express®

Programming WebLogic JDBC

Copyright

Copyright © 2002 - 2006 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

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

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

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

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Programming WebLogic JDBC

Part Number	Date	Software Version
N/A	April 8, 2004	BEA WebLogic Server Version 7.0

Contents

About This Document

Audience.....	xii
e-docs Web Site.....	xii
How to Print the Document.....	xii
Related Information.....	xiii
Contact Us!	xiii
Documentation Conventions	xiv

1. Introduction to WebLogic JDBC

Overview of JDBC	1-1
Using JDBC Drivers with WebLogic Server	1-2
Types of JDBC Drivers	1-2
Table of WebLogic Server JDBC Drivers	1-2
WebLogic Server JDBC Two-Tier Drivers	1-3
WebLogic jDriver for Oracle	1-4
WebLogic jDriver for Microsoft SQL Server	1-4
WebLogic Server JDBC Multitier Drivers.....	1-4
WebLogic RMI Driver.....	1-4
WebLogic Pool Driver	1-5
WebLogic JTS Driver	1-5
Third-Party Drivers	1-5
Sybase jConnect Driver	1-6
Oracle Thin Driver	1-6
Overview of Connection Pools.....	1-6
Using Connection Pools with Server-side Applications	1-8
Using Connection Pools with Client-side Applications.....	1-9
Overview of MultiPools	1-9

Overview of Clustered JDBC	1-10
Overview of DataSources	1-10
JDBC API	1-10
JDBC 2.0	1-11
Platforms	1-11

2. Configuring and Administering WebLogic JDBC

Configuring and Using Connection Pools	2-2
Advantages to Using Connection Pools	2-2
Creating a Connection Pool at Startup	2-3
Avoiding Server Lockup with the Correct Number of Connections...	2-3
Database Passwords in Connection Pool Configuration	2-3
Connection Pool Limitation	2-5
Notes About Refreshing Connections in a JDBC Connection Pool....	2-5
JDBC Connection Pool Testing Enhancements	2-6
Minimizing Connection Test Delay After Database Connectivity Loss..	2-6
Minimizing Connection Request Delay After Connection Test Failures	2-7
Minimizing Connection Request Delay with	
SecondsToTrustAnIdlePoolConnection.....	2-9
Creating a Connection Pool Dynamically	2-10
Dynamic Connection Pool Sample Code	2-11
Import Packages	2-11
Look Up the Administration MBeanHome	2-11
Get the Server MBean	2-12
Create the Connection Pool MBean	2-12
Set the Connection Pool Properties	2-12
Add the Target.....	2-13
Create a DataSource	2-13
Removing a Dynamic Connection Pool and DataSource.....	2-13
Managing Connection Pools	2-14
Retrieving Information About a Pool.....	2-15
Disabling a Connection Pool.....	2-15
Shrinking a Connection Pool.....	2-16
Shutting Down a Connection Pool.....	2-16

Resetting a Pool	2-17
Using weblogic.jdbc.common.JdbcServices and weblogic.jdbc.common.Pool Classes (Deprecated)	2-18
Application-Scoped JDBC Connection Pools	2-19
Configuring and Using MultiPools	2-20
MultiPool Features	2-20
Choosing the MultiPool Algorithm.....	2-20
High Availability	2-21
Load Balancing	2-21
MultiPool Failover Enhancements	2-21
Connection Request Routing Enhancements When a Connection Pool Fails.....	2-22
Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool	2-22
Enabling Failover for Busy Connection Pools in a MultiPool.....	2-23
Controlling MultiPool Failover with a Callback.....	2-24
Controlling MultiPool Failback with a Callback	2-27
MultiPool Fail-Over Limitations and Requirements.....	2-29
Test Connections on Reserve to Enable Fail-Over	2-29
No Fail-Over for In-Use Connections.....	2-30
Configuring and Using DataSources	2-30
Importing Packages to Access DataSource Objects.....	2-31
Obtaining a Client Connection Using a DataSource	2-31
Code Examples	2-32
JDBC Data Source Factories.....	2-32

3. Performance Tuning Your JDBC Application

Overview of JDBC Performance.....	3-1
WebLogic Performance-Enhancing Features.....	3-1
How Connection Pools Enhance Performance.....	3-2
Caching Prepared Statements and Data	3-2
Designing Your Application for Best Performance	3-3
1. Process as Much Data as Possible Inside the Database	3-3
2. Use Built-in DBMS Set-based Processing	3-4
3. Make Your Queries Smart.....	3-4
4. Make Transactions Single-batch	3-6

5. Never Have a DBMS Transaction Span User Input.....	3-7
6. Use In-place Updates.....	3-7
7. Keep Operational Data Sets Small	3-8
8. Use Pipelining and Parallelism.....	3-8

4. Using WebLogic Multitier JDBC Drivers

Using the WebLogic RMI Driver	4-1
Setting Up WebLogic Server to Use the WebLogic RMI Driver	4-2
Sample Client Code for Using the RMI Driver.....	4-2
Import the Required Packages.....	4-2
Get the Database Connection	4-3
Using a JNDI Lookup to Obtain the Connection	4-3
Using Only the WebLogic RMI Driver to Obtain a Database Connection	
4-4	
Row Caching with the WebLogic RMI Driver	4-5
Important Limitations for Row Caching with the WebLogic RMI Driver	
4-6	
Using the WebLogic JTS Driver	4-7
Sample Client Code for Using the JTS Driver	4-7
Using the WebLogic Pool Driver	4-9

5. Using Third-Party Drivers with WebLogic Server

Overview of Third-Party JDBC Drivers.....	5-1
Setting the Environment for Your Third-Party JDBC Driver	5-4
CLASSPATH for Third-Party JDBC Driver on Windows	5-4
CLASSPATH for Third-Party JDBC Driver on UNIX.....	5-4
Changing or Updating the Oracle Thin Driver.....	5-5
Package Change for Oracle Thin Driver 9.x and 10g	5-6
Character Set Support with nls_charset12.zip.....	5-6
Updating Sybase jConnect Driver	5-7
Installing and Using the IBM Informix JDBC Driver.....	5-7
Connection Pool Attributes when using the IBM Informix JDBC Driver	
5-8	
Programming Notes for the IBM Informix JDBC Driver.....	5-10
Installing and Using the SQL Server 2000 Driver for JDBC from Microsoft	
5-10	

Installing the MS SQL Server JDBC Driver on a Windows System	5-11
Installing the MS SQL Server JDBC Driver on a Unix System	5-11
Connection Pool Attributes when using the Microsoft SQL Server Driver for JDBC	5-12
Getting a Connection with Your Third-Party Driver	5-13
Using Connection Pools with a Third-Party Driver	5-13
Creating the Connection Pool and DataSource	5-14
Using a JNDI Lookup to Obtain the Connection	5-14
Getting a Physical Connection from a Connection Pool	5-15
Code Sample for Getting a Physical Connection	5-16
Limitations for Using a Physical Connection	5-18
Using Oracle Extensions with the Oracle Thin Driver	5-18
Limitations When Using Oracle JDBC Extensions	5-19
Sample Code for Accessing Oracle Extensions to JDBC Interfaces	5-19
Import Packages to Access Oracle Extensions	5-20
Establish the Connection	5-20
Retrieve the Default Row Prefetch Value	5-21
Programming with ARRAYS	5-21
Getting an ARRAY	5-22
Updating ARRAYS in the Database	5-23
Using Oracle Array Extension Methods	5-23
Programming with STRUCTs	5-24
Getting a STRUCT	5-25
Using OracleStruct Extension Methods	5-25
Getting STRUCT Attributes	5-26
Using STRUCTs to Update Objects in the Database	5-27
Creating Objects in the Database	5-27
Automatic Buffering for STRUCT Attributes	5-28
Programming with REFs	5-29
Getting a REF	5-29
Using OracleRef Extension Methods	5-30
Getting a Value	5-30
Updating REF Values	5-31
Creating a REF in the Database	5-33
Programming with BLOBs and CLOBs	5-34

Query to Select BLOB Locator from the DBMS	5-34
Declare the WebLogic Server java.sql Objects.....	5-34
Begin SQL Exception Block	5-34
Updating a CLOB Value Using a Prepared Statement	5-35
Programming with Oracle Virtual Private Databases.....	5-35
Tables of Oracle Extension Interfaces and Supported Methods.....	5-36

6. Using dbKona (Deprecated)

Overview of dbKona	6-1
dbKona in a Multitier Configuration.....	6-2
How dbKona and a JDBC Driver Interact.....	6-2
How dbKona and WebLogic Events Can interact.....	6-3
The dbKona Architecture	6-3
The dbKona API.....	6-4
The dbKona API Reference.....	6-4
The dbKona Objects and Their Classes.....	6-5
Data Container Objects in dbKona.....	6-5
DataSet	6-5
QueryDataSet	6-6
TableDataSet	6-7
EventfulTableDataSet (Deprecated)	6-9
Record	6-10
Value	6-11
Data Description Objects in dbKona.....	6-12
Schema	6-12
Column	6-13
KeyDef	6-13
SelectStmt.....	6-14
Miscellaneous Objects in dbKona.....	6-14
Exceptions	6-15
Constants	6-15
Entity Relationships.....	6-15
Inheritance Relationships	6-15
Possession Relationships.....	6-16
Implementing dbKona	6-16

Accessing a DBMS with dbKona.....	6-16
Step 1. Import packages	6-17
Step 2. Set Properties for Making a Connection	6-17
Step 3. Make a Connection to the DBMS.....	6-17
Preparing a Query, Retrieving, and Displaying Data.....	6-18
Step 1. Set Parameters for Data Retrieval.....	6-18
Step 2. Create a DataSet for the Query Results	6-19
Step 3. Fetch the Results	6-20
Step 4. Examine a TableDataSet's Schema	6-21
Step 5. Examine the Data with htmlKona.....	6-21
Step 6. Display the Results with htmlKona	6-22
Step 7. Close the DataSet and the Connection.....	6-22
Using a SelectStmt Object to Form a Query	6-25
Step 1. Setting SelectStmt Parameters	6-25
Step 2. Using QBE to Refine the Parameters.....	6-26
Modifying DBMS Data with a SQL Statement	6-26
Step 1. Writing SQL Statements	6-26
Step 1. Writing SQL statements.....	6-27
Step 2. Executing Each SQL Statement.....	6-27
Step 3. Displaying the Results with htmlKona	6-27
Modifying DBMS Data with a KeyDef	6-31
Step 1. Creating a KeyDef and Building Its Attributes.....	6-31
Step 2. Creating a TableDataSet with a KeyDef.....	6-31
Step 3. Inserting a Record into the TableDataSet	6-32
Step 4. Updating a Record in the TableDataSet.....	6-32
Step 5. Deleting a Record from the TableDataSet.....	6-33
Step 6. More on Saving the TableDataSet	6-33
Checking Record Status Before Saving.....	6-33
Step 7. Verifying the changes	6-34
Code Summary.....	6-35
Using a JDBC PreparedStatement with dbKona.....	6-36
Using Stored Procedures with dbKona	6-37
Step 1. Creating a Stored Procedure	6-38
Step 2. Setting parameters.....	6-38
Step 3. Examining the Results	6-38

Using Byte Arrays for Images and Audio	6-39
Step 1. Retrieving and Displaying Image Data	6-39
Step 2. Inserting an Image into a Database	6-40
Using dbKona for Oracle Sequences	6-40
Constructing a dbKona Sequence Object	6-40
Creating and Destroying Sequences on an Oracle Server from dbKona .	
6-41	
Using a Sequence	6-41
Code Summary	6-41

7. Testing JDBC Connections and Troubleshooting

Monitoring JDBC Connectivity	7-1
Validating a DBMS Connection from the Command Line	7-2
Syntax	7-2
Arguments	7-3
Examples	7-3
Troubleshooting JDBC	7-4
JDBC Connections	7-4
Windows	7-4
UNIX	7-5
Codeset Support	7-5
Other Problems with Oracle on UNIX	7-5
Thread-related Problems on UNIX	7-5
Closing JDBC Objects	7-6
Abandoning JDBC Objects	7-7
Troubleshooting Problems with Shared Libraries on UNIX	7-8
WebLogic jDriver for Oracle	7-8
Solaris	7-8
HP-UX	7-9
Incorrectly Set File Permissions	7-9
Incorrect SHLIB_PATH	7-10
Using Microsoft SQL with Nested Triggers	7-10
Exceeding the Nesting Level	7-10
Using Triggers and EJBs	7-11

About This Document

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

The document is organized as follows:

- [Chapter 1, “Introduction to WebLogic JDBC,”](#) introduces the JDBC components and JDBC API.
- [Chapter 2, “Configuring and Administering WebLogic JDBC,”](#) describes how to configure JDBC components for use with WebLogic Server Java applications.
- [Chapter 3, “Performance Tuning Your JDBC Application,”](#) describes how to obtain the best performance from JDBC applications.
- [Chapter 4, “Using WebLogic Multitier JDBC Drivers,”](#) describes how to set up your WebLogic RMI driver and JDBC clients to use with WebLogic Server.
- [Chapter 5, “Using Third-Party Drivers with WebLogic Server,”](#) describes how to set up and use third-party drivers with WebLogic Server.
- [Chapter 6, “Using dbKona \(Deprecated\),”](#) describes how to use dbKona classes in your applications.
- [Chapter 7, “Testing JDBC Connections and Troubleshooting,”](#) describes troubleshooting tips when using JDBC with WebLogic Server.

Audience

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

Contact Us!

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
<i>italics</i>	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. <i>Examples:</i> <pre>import java.util.Enumeration; chmod u+w * config/examples/applications .java config.xml float</pre>
<i>monospace italic text</i>	Variables in code. <i>Example:</i> <pre>String CustomerName;</pre>
UPPERCASE TEXT	Device names, environment variables, and logical operators. <i>Examples:</i> <pre>LPT1 BEA_HOME OR</pre>
{ }	A set of choices in a syntax line.
[]	Optional items in a syntax line. <i>Example:</i> <pre>java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]</pre>

Convention	Usage
	Separates mutually exclusive choices in a syntax line. <i>Example:</i> <pre>java weblogic.deploy [list deploy undeploy update] password {application} {source}</pre>
...	Indicates one of the following in a command line: <ul style="list-style-type: none">■ 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.



1 Introduction to WebLogic JDBC

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

- [“Overview of JDBC” on page 1-1](#)
- [“Using JDBC Drivers with WebLogic Server” on page 1-2](#)
- [“Overview of Connection Pools” on page 1-6](#)
- [“Overview of MultiPools” on page 1-9](#)
- [“Overview of Clustered JDBC” on page 1-10](#)
- [“Overview of DataSources” on page 1-10](#)
- [“JDBC API” on page 1-10](#)
- [“JDBC 2.0” on page 1-11](#)
- [“Platforms” on page 1-11](#)

Overview of JDBC

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

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

Using JDBC Drivers with WebLogic Server

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

Types of JDBC Drivers

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

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

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

Table of WebLogic Server JDBC Drivers

The following table summarizes the drivers that WebLogic Server uses.

Table 1-1 JDBC Drivers

Driver Tier	Type and Name of Driver	Database Connectivity	Documentation Sources
Two-tier without support for distributed transactions (non-XA)	Type 2 (requires native libraries): <ul style="list-style-type: none"> ■ WebLogic jDriver for Oracle ■ Third-party drivers Type 4 (pure Java) <ul style="list-style-type: none"> ■ WebLogic jDrivers for Microsoft SQL Server ■ Third-party drivers, including: Oracle Thin Sybase jConnect 	Between WebLogic Server and DBMS in local transactions.	<i>Programming WebLogic JDBC</i> (this document) <i>Administration Guide</i> , “ Managing JDBC Connectivity ” <i>Using WebLogic jDriver for Oracle</i> <i>Using WebLogic jDriver for Microsoft SQL Server</i>
Two-tier with support for distributed transactions (XA)	Type 2 (requires native libraries) <ul style="list-style-type: none"> ■ WebLogic jDriver for Oracle XA 	Between WebLogic Server and DBMS in distributed transactions.	<i>Programming WebLogic JTA Administration Guide</i> , “ Managing JDBC Connectivity ” <i>Using WebLogic jDriver for Oracle</i>
Multitier	Type 3 <ul style="list-style-type: none"> ■ WebLogic RMI Driver ■ WebLogic Pool Driver ■ WebLogic JTS (not Type 3) 	Between client and WebLogic Server (connection pool). The RMI driver replaces the deprecated t3 driver. The JTS driver is used in distributed transactions. The Pool and JTS drivers are server-side only.	<i>Programming WebLogic JDBC</i> (this document)

WebLogic Server JDBC Two-Tier Drivers

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

WebLogic jDriver for Oracle

BEA's WebLogic jDriver for Oracle is included with the WebLogic Server distribution. This driver requires an Oracle client installation. The *WebLogic jDriver for Oracle XA* driver extends the WebLogic jDriver for Oracle for distributed transactions. For additional information, see [Using WebLogic jDriver for Oracle at `http://e-docs.bea.com/wls/docs70oracle/index.html`](http://e-docs.bea.com/wls/docs70oracle/index.html).

WebLogic jDriver for Microsoft SQL Server

BEA's WebLogic jDriver for Microsoft SQL Server, included in the WebLogic Server distribution, is a pure-Java, Type 4 JDBC driver that provides connectivity to Microsoft SQL Server. For more information, see [Configuring and Using WebLogic jDriver for MS SQL Server at `http://e-docs.bea.com/wls/docs70/mssqlserver4/index.html`](http://e-docs.bea.com/wls/docs70/mssqlserver4/index.html).

WebLogic Server JDBC Multitier Drivers

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

For more details about using these drivers, see [Chapter 4, "Using WebLogic Multitier JDBC Drivers."](#)

WebLogic RMI Driver

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

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

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

For more details about using the WebLogic RMI driver, see [“Using the WebLogic RMI Driver” on page 4-1](#).

WebLogic Pool Driver

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

For information about using the Pool driver, see Accessing Databases in [Programming Tasks](#) in *Programming WebLogic HTTP Servlets*.

WebLogic JTS Driver

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

For more details about using the WebLogic JTS driver, see [“Using the WebLogic JTS Driver” on page 4-7](#).

Third-Party Drivers

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

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

- Implement EJB transaction calls in JDBC.

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

Sybase jConnect Driver

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

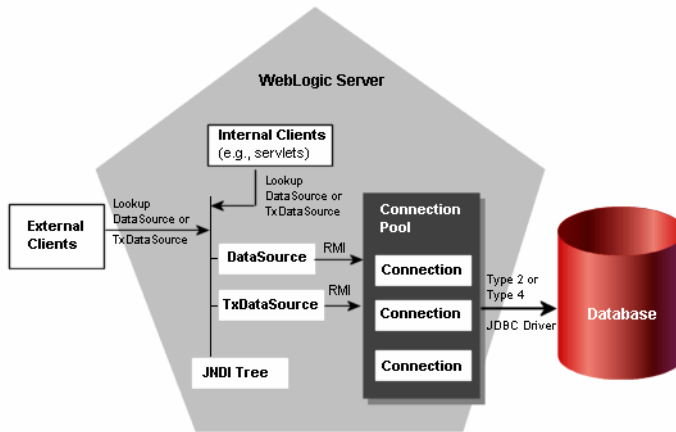
Oracle Thin Driver

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

Overview of Connection Pools

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

Figure 1-1 WebLogic Server Connection Pool Architecture



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

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

Table 1-2 Advantages to Using Connection Pools

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

This section is an overview of connection pools. For more detailed information, see [“Configuring and Using Connection Pools” on page 2-2](#).

Using Connection Pools with Server-side Applications

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

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

Using Connection Pools with Client-side Applications

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

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

Overview of MultiPools

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

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

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

For more information, see [“Configuring and Using MultiPools” on page 2-20](#).

Overview of Clustered JDBC

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

For information about JDBC objects in a clustered environment, see “[JDBC Connections](http://e-docs.bea.com/wls/docs70/cluster/overview.html#jdbc_connections)” in *Using WebLogic Server Clusters* at http://e-docs.bea.com/wls/docs70/cluster/overview.html#jdbc_connections.

Overview of DataSources

Client and server-side JDBC applications can obtain a DBMS connection using a DataSource. A DataSource is an interface between an application and the connection pool. Each data source (such as a DBMS instance) requires a separate DataSource object, which may be implemented as a DataSource class that supports distributed transactions. For more information, see “[Configuring and Using DataSources](#)” on page 2-30.

JDBC API

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

- [weblogic.jdbc.pool](#)

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

JDBC 2.0

WebLogic Server supports JDBC 2.0.

Platforms

Supported platforms vary by vendor-specific DBMSs and drivers. For current information, see [BEA WebLogic Server Platform Support at `http://e-docs.bea.com/platform/suppconfigs/index.html`](#).

2 Configuring and Administering WebLogic JDBC

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

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

- “Configuring and Using Connection Pools” on page 2-2
- “Application-Scoped JDBC Connection Pools” on page 2-19
- “Configuring and Using MultiPools” on page 2-20
- “Configuring and Using DataSources” on page 2-30

For additional information, see

- **Managing JDBC Connectivity** in the *Administration Guide* at <http://e-docs.bea.com/wls/docs70/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/docs70/ConsoleHelp/index.html>. Describes how to use the Administration Console to set specific configuration tasks.

Configuring and Using Connection Pools

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

Advantages to Using Connection Pools

Connection pools provide numerous performance and application design advantages:

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

The attributes for configuring a connection pool are defined in the [Administration Console Online Help](#). There is also an API that you can use to programmatically create connection pools in a running WebLogic Server; see [“Creating a Connection Pool Dynamically” on page 2-10](#). 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/docs70/adminguide/cli.html>.

Creating a Connection Pool at Startup

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

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

For step-by-step instructions and a description of connection pool attributes, see the [Administration Console Online Help](http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html), available when you click the question mark in the upper-right corner of the Administration Console [or at](#) <http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html>. For more information about creating and configuring connection pools with the Administration Console, see “[Managing JDBC Connectivity](#)” in the *Administration Guide* at <http://e-docs.bea.com/wls/docs70/adminguide/jdbc.html>.

Avoiding Server Lockup with the Correct Number of Connections

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

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

Database Passwords in Connection Pool Configuration

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

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

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

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

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

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

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

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


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

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

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.

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.

JDBC Connection Pool Testing Enhancements

In WebLogic Server 7.0SP5, the following attributes were added to JDBC connection pools to improve the functionality of database connection testing for pooled connections:

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

Minimizing Connection Test Delay After Database Connectivity Loss

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

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

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

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

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

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

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

Minimizing Connection Request Delay After Connection Test Failures

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

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

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

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

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

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

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

Minimizing Connection Request Delay with SecondsToTrustAnIdlePoolConnection

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

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

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

To set `secondsToTrustAnIdlePoolConnection`, you add it to the list of Properties on the JDBC Connection Pool → Configuration → General tab in the Administration Console. See "[JDBC Connection Pool --> Configuration --> General](#)" in the *Administration Console Online Help*. You can also set it directly in the `config.xml` file. For example:

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    secondsToTrustAnIdlePoolConnection=15;
  DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  TestConnectionsOnReserve="true"
  TestTableName="SYSTABLES"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

`SecondsToTrustAnIdlePoolConnection` is a tuning feature that can improve application performance by minimizing the delay caused by database connection testing, especially during heavy traffic. However, it can reduce the effectiveness of

connection testing, especially if the value is set too high. The appropriate value depends on your environment and the likelihood that a connection will become defunct.

Creating a Connection Pool Dynamically

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

You can also use the `CREATE_POOL` command in the WebLogic Server command line interface to dynamically create a connection pool. See [CREATE_POOL](http://e-docs.bea.com/wls/docs70/adminguide/cli.html#cli_create_pool) in the *Administration Guide* at http://e-docs.bea.com/wls/docs70/adminguide/cli.html#cli_create_pool.

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

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

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

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

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

For more information about using MBeans to manage WebLogic Server, see [Programming WebLogic Management Services with JMX at `http://e-docs.bea.com/wls/docs70/jmx/index.html`](http://e-docs.bea.com/wls/docs70/jmx/index.html). For more information about the `JDBCConnectionPool` MBean, see the [Javadoc for WebLogic Classes at `http://e-docs.bea.com/wls/docs70/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html`](http://e-docs.bea.com/wls/docs70/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html).

Dynamic Connection Pool Sample Code

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

Import Packages

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

Look Up the Administration MBeanHome

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

Get the Server MBean

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

Create the Connection Pool MBean

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

Set the Connection Pool Properties

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

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

Note: In this example, the database password is set using the `setPassword(String)` method instead of including it with the user and server names in `Properties`. When you use the `setPassword(String)` method, WebLogic Server encrypts the password in the `config.xml` file and

when displayed on the administration console. BEA recommends that you use this method to avoid storing database passwords in clear text in the `config.xml` file.

Add the Target

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

Create a DataSource

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

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

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

        // Startup datasource
        dsMBeans.addTarget(serverMBean);

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

Removing a Dynamic Connection Pool and DataSource

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

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

public void deleteDataSource() throws SQLException {
    try {
        // Remove dynamically created datasource from the server

        dsMBeans.removeTarget(serverMBean);

        // Remove dynamically created datasource from the configuration

        mbeanHome.deleteMBean(dsMBeans);
    } catch (Exception ex) {
        throw new SQLException(ex.toString());
    }
}
```

Managing Connection Pools

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

- Retrieving information about a pool
- Disabling a connection pool, which prevents clients from obtaining a connection from it
- Enabling a disabled pool

- Shrinking a pool, which releases unused connections until the pool has reached the minimum specified pool size
- Refreshing a pool, which closes and reopens its connections
- Shutting down a pool

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

For more information about methods provided by the `JDBCConnectionPool` MBean, see the [Javadoc at](#)

<http://e-docs.bea.com/wls/docs70/javadocs/weblogic/management/configuration/JDBCConnectionPoolMBean.html>. For more information about the methods provided by the `JDBCConnectionPoolRuntime` MBean, see the [Javadoc at](#) <http://e-docs.bea.com/wls/docs70/javadocs/weblogic/management/runtime/JDBCConnectionPoolRuntimeMBean.html>.

Retrieving Information About a Pool

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

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

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

Disabling a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.disableDroppingUsers()  
  
JDBCConnectionPoolRuntimeMBean.disableFreezingUsers()  
  
JDBCConnectionPoolRuntimeMBean.enable()
```

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

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

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

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

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

Shrinking a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.shrink()
```

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

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

Shutting Down a Connection Pool

```
JDBCConnectionPoolRuntimeMBean.shutdownSoft()
```

```
JDBCConnectionPoolRuntimeMBean.shutdownHard()
```

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

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

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

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

Resetting a Pool

```
JDBCConnectionPoolRuntimeMBean.reset()
```

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

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

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

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

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

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

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

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

- a. In a `try` block, test a connection from the connection pool with a SQL statement that is guaranteed to succeed under any circumstances so long as there is a working connection to the DBMS. An example is the SQL statement `select 1 from dual` which is guaranteed to succeed for an Oracle DBMS.
- b. Catch the `SQLException`.
- c. Call the `reset()` method in the catch block.

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

Previous versions of WebLogic Server included classes that you could use to programmatically create and manage connection pools:

`weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool`.

These classes are now deprecated. Although these classes are still available, BEA recommends that you use the `JDBCConnectionPool` MBean instead of these classes to dynamically create and manage connection pools.

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

- `shutdown`
- `retrieve`
- `refresh`
- `enable`
- `disable`

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

For more information about `weblogic.jdbc.common.JdbcServices` and `weblogic.jdbc.common.Pool`, see “[Configuring WebLogic JDBC Features](#)” in *Programming WebLogic JDBC* for WebLogic Server 6.1 at

<http://edocs.bea.com/wls/docs61/jdbc/programming.html>.

Application-Scoped JDBC Connection Pools

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

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

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

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

- [weblogic-application.xml Deployment Descriptor Elements](http://e-docs.bea.com/wls/docs70/programming/app_xml.html#app-scoped-pool) in *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs70/programming/app_xml.html#app-scoped-pool.
- [Packaging Enterprise Applications](http://e-docs.bea.com/wls/docs70/programming/packaging.html#pack009) in *Developing WebLogic Server Applications* at <http://e-docs.bea.com/wls/docs70/programming/packaging.html#pack009>.
- [Two-Phase Deployment](http://e-docs.bea.com/wls/docs70/programming/deploying.html#two-phasedeploy) in *Developing WebLogic Server Applications* at <http://e-docs.bea.com/wls/docs70/programming/deploying.html#two-phasedeploy>.

Configuring and Using MultiPools

A MultiPool is a “pool of pools.” You create a MultiPool by first creating connection pools, then creating the MultiPool using the Administration Console or WebLogic Management API and assigning the connection pools to the MultiPool.

For instructions to create a MultiPool using the Administration Console, see the [Administration Console Online Help](#) at

http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_metapool_create. For information about the `JDBCMultiPoolMBean`, see the [WebLogic Server Javadocs](#) at

<http://e-docs.bea.com/wls/docs70/javadocs/weblogic/management/configuration/JDBCMultiPoolMBean.html>.

Note: If you are not using global transactions (XA), you can only use MultiPools to connect to Oracle RAC.

MultiPool Features

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

Database connections from a MultiPool are used in local transactions only and are not supported by WebLogic Server for use in distributed transactions.

Choosing the MultiPool Algorithm

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

High Availability

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

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.

By default, 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. You can enable failover in this scenario by setting the `FailoverRequestIfBusy` attribute in the MultiPool configuration to `true`. See [“Enabling Failover for Busy Connection Pools in a MultiPool” on page 2-23](#) for more details.

Load Balancing

Connection requests to a load balancing MultiPool are served from any connection pool in the list. Pools are added in the order listed and are accessed using a round-robin scheme. When an application requests a connection, the MultiPool attempts to provide a connection from the next connection pool in the list.

MultiPool Failover Enhancements

In WebLogic Server 7.0SP5, the following enhancements were made to MultiPools:

- Connection request routing enhancements to avoid requesting a connection from an automatically disabled (dead) connection pool within a MultiPool. See [“Connection Request Routing Enhancements When a Connection Pool Fails.”](#)
- Automatic failback on recovery of a failed connection pool within a MultiPool. See [“Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool.”](#)
- Failover for busy connection pools within a MultiPools. See [“Enabling Failover for Busy Connection Pools in a MultiPool.”](#)

- Failover callbacks for MultiPools with the High Availability algorithm. See [“Controlling MultiPool Failover with a Callback.”](#)
- Failback callbacks for MultiPools with either algorithm. See [“Controlling MultiPool Failback with a Callback.”](#)

Connection Request Routing Enhancements When a Connection Pool Fails

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

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

If a callback handler is registered for the MultiPool, WebLogic Server calls the callback handler before failing over to the next connection pool in the list. See [“Controlling MultiPool Failover with a Callback”](#) on page 2-24 for more details.

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

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

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

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
PoolList="demoPool2,demoPool"
```

```
HealthCheckFrequencySeconds="240"  
Targets="examplesServer" />
```

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

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

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

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

If a callback handler is registered for the MultiPool, WebLogic Server calls the callback handler before re-enabling the connection pool. See [“Controlling MultiPool Failback with a Callback” on page 2-27](#) for more details.

Enabling Failover for Busy Connection Pools in a MultiPool

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

To enable the MultiPool to failover when all connections in the current connection pool are in use, you must set a value for the `FailoverRequestIfBusy` attribute in the MultiPool configuration in the `config.xml` file. If set to `true`, when all connections in the current connection pool are in use, application requests for connections will be routed to the next available connection pool within the MultiPool. When set to `false` (the default), connection requests do not failover. [Which exception is thrown?]

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

```
<JDBCMultiPool  
AlgorithmType="High-Availability"  
Name="demoMultiPool"  
PoolList="demoPool2,demoPool"
```

```
FailoverRequestIfBusy="true"  
Targets="examplesServer" />
```

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

If a `ConnectionPoolFailoverCallbackHandler` is included in the MultiPool configuration, WebLogic Server calls the callback handler before failing over. See [“Controlling MultiPool Failover with a Callback” on page 2-24](#) for more details.

Controlling MultiPool Failover with a Callback

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

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

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

Callback Handler Requirements

A callback handler used to control the failover and failback within a MultiPool must include an implementation of the

`weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface.

When the MultiPool needs to failover to the next connection pool in the list or when a previously disabled connection pool becomes available, WebLogic Server calls the `allowPoolFailover()` method in the `ConnectionPoolFailoverCallback` interface, and passes a value for the three parameters, `currPool`, `nextPool`, and `opcode`, as defined below. WebLogic Server then waits for the return from the callback handler before completing the task.

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

See the Javadoc for the

`weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface for more details. [\[Add link\]](#)

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

Callback Handler Configuration

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

- `ConnectionPoolFailoverCallbackHandler`—To register a failover callback handler for a MultiPool, you add a value for this attribute to the MultiPool configuration in the `config.xml` file. The value must be an absolute name, such as `com.bea.samples.wls.jdbc.MultiPoolFailoverCallbackApplication`.
- `HealthCheckFrequencySeconds`—To control how often WebLogic Server checks disabled (dead) connection pools in a MultiPool to see if they are now available, you can add a value for this attribute to the MultiPool configuration in the `config.xml` file. See [“Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool” on page 2-22](#) for more details.

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

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

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

How It Works—Failover

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

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

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

- `currPool`—For failover, this is the name of connection pool currently being used to supply database connections. This is the “failover from” connection pool.
- `nextPool`—The name of next available connection pool listed in the `MultiPool`. For failover, this is the “failover to” connection pool.
- `opcode`—A code that indicates the reason for the call:
 - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current connection pool is dead and has disabled it.
 - `OPCODE_CURR_POOL_BUSY`—All database connections in the connection pool are in use. (Requires `FailoverIfBusy=true` in the `MultiPool` configuration. See [“Enabling Failover for Busy Connection Pools in a MultiPool” on page 2-23.](#))

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

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

- `OK`—proceed with the operation. In this case, that means to failover to the next connection pool in the list.
- `RETRY_CURRENT`—Retry the connection request with the current connection pool.

- `DONOT_FAILOVER`—Do not retry the current connection request and do not failover. WebLogic Server will throw a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

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

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

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

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

Controlling MultiPool Failback with a Callback

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

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

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

- [“Callback Handler Requirements” on page 2-24](#)
- [“Callback Handler Configuration” on page 2-25](#)

How It Works—Failback

WebLogic Server periodically checks the status of connection pools in a MultiPool that were automatically disabled. (See [“Automatic Re-enablement on Recovery of a Failed Connection Pool within a MultiPool” on page 2-22.](#)) If a disabled connection pool becomes available and if a failover callback handler is registered, WebLogic Server calls the callback handler with the following information and waits for a return:

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

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

The callback handler can return one of the following values:

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

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

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

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

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

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

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

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

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

MultiPool Fail-Over Limitations and Requirements

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

Test Connections on Reserve to Enable Fail-Over

Connection pools rely on the `TestConnectionsOnReserve` feature to know when database connectivity is lost. Connections are *not* automatically tested before being reserved by an application. You must set `TestConnectionsOnReserve=true` for the connection pools within the `MultiPool`. After you turn 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. See [“MultiPool Failover Enhancements” on page 2-21](#) for details about enhancements to `MultiPool` failover.

No Fail-Over for In-Use Connections

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

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/docs70/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](http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_data_source_create) at

http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_data_source_create. For instructions to create a TxDataSource, see the [Administration Console Online Help](http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_tx_data_source_create) at http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_tx_data_source_create.

Importing Packages to Access DataSource Objects

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

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

Obtaining a Client Connection Using a DataSource

To obtain a connection from a JDBC client, use a Java Naming and Directory Interface (JNDI) 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:
```

2 Configuring and Administering WebLogic JDBC

```
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](http://e-docs.bea.com/wls/docs70/jndi/index.html) at <http://e-docs.bea.com/wls/docs70/jndi/index.html>.

Code Examples

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

JDBC Data Source Factories

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

For details about creating and using a JDBC Data Source factory, see [Resource Factories](http://e-docs.bea.com/wls/docs70/ejb/EJB_environment.html#resourcefact) in *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/EJB_environment.html#resourcefact.

3 Performance Tuning Your JDBC Application

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

- [“Overview of JDBC Performance” on page 3-1](#)
- [“WebLogic Performance-Enhancing Features” on page 3-1](#)
- [“Designing Your Application for Best Performance” on page 3-3](#)

Overview of JDBC Performance

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

WebLogic Performance-Enhancing Features

WebLogic has several features that enhance performance for JDBC applications.

How Connection Pools Enhance Performance

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

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

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

Caching Prepared Statements and Data

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

- **Prepared Statement Cache** for a connection pool
(<http://e-docs.bea.com/wls/docs70/adminguide/jdbc.html#preparedstatementcache>)
- **Read-Only Entity Beans**
(http://e-docs.bea.com/wls/docs70/ejb/EJB_environment.html)
- **JNDI in a Clustered Environment**
(<http://e-docs.bea.com/wls/docs70/jndi/jndi.html>)

Designing Your Application for Best Performance

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

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

1. Process as Much Data as Possible Inside the Database

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

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

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

2. Use Built-in DBMS Set-based Processing

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

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

3. Make Your Queries Smart

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

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

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

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

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

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

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

```
select * from payroll
```

Table 3-1 Full Results Returned

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

A correlated subquery

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

returns a much smaller result, shown in [Table 3-2](#).

Table 3-2 Results from Subquery

Name	Salary
Sue	70
Hal	80
May	80

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

4. Make Transactions Single-batch

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

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

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

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

5. Never Have a DBMS Transaction Span User Input

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

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

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

6. Use In-place Updates

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

7. Keep Operational Data Sets Small

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

8. Use Pipelining and Parallelism

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

4 Using WebLogic Multitier JDBC Drivers

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

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

- [“Using the WebLogic RMI Driver” on page 4-1](#)
- [“Using the WebLogic JTS Driver” on page 4-7](#)
- [“Using the WebLogic Pool Driver” on page 4-9](#)

Using the WebLogic RMI Driver

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

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

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

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

Setting Up WebLogic Server to Use the WebLogic RMI Driver

The RMI driver is accessible only through DataSource objects, which are created in the Administration Console. You must create DataSource objects in your WebLogic Server configuration before you can use the RMI driver in your applications. For instructions to create a DataSource, see the [Administration Console Online Help at `http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_data_source_create`](http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_data_source_create). For instructions to create a TxDataSource, see the [Administration Console Online Help at `http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_tx_data_source_create`](http://e-docs.bea.com/wls/docs70/ConsoleHelp/jdbc.html#jdbc_tx_data_source_create).

Sample Client Code for Using the RMI Driver

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

Import the Required Packages

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

```
javax.sql.DataSource
java.sql.*
```

```
java.util.*
javax.naming.*
```

Get the Database Connection

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

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

Using a JNDI Lookup to Obtain the Connection

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

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

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

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

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

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

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

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

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see [Programming WebLogic JNDI at
http://e-docs.bea.com/wls/docs70/jndi/index.html](http://e-docs.bea.com/wls/docs70/jndi/index.html).

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

Using Only the WebLogic RMI Driver to Obtain a Database Connection

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

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

String url ="jdbc:weblogic:rmi";

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

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

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

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

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

Row Caching with the WebLogic RMI Driver

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

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

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

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

Important Limitations for Row Caching with the WebLogic RMI Driver

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

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

- `getDate()`
- `getTime()`
- `getTimestamp()`

Using the WebLogic JTS Driver

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

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

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

Sample Client Code for Using the JTS Driver

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

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

1. Import the following classes:

```
import javax.transaction.UserTransaction;
import java.sql.*;
import javax.naming.*;
import java.util.*;
import weblogic.jndi.*;
```

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

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

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

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

ctx = new InitialContext(env);

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

3. Start a transaction on the current thread:

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

4. Load the JTS driver:

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

5. Get a connection from the connection pool:

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

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

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

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

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

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

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

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

```
tx.commit();
```

```
// or:
```

```
tx.rollback();
```

Using the WebLogic Pool Driver

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

5 Using Third-Party Drivers with WebLogic Server

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

- [“Overview of Third-Party JDBC Drivers” on page 5-1](#)
- [“Setting the Environment for Your Third-Party JDBC Driver” on page 5-4](#)
- [“Getting a Connection with Your Third-Party Driver” on page 5-13](#)
- [“Using Oracle Extensions with the Oracle Thin Driver” on page 5-18](#)
- [“Programming with Oracle Virtual Private Databases” on page 5-35](#)
- [“Tables of Oracle Extension Interfaces and Supported Methods” on page 5-36](#)

Overview of Third-Party JDBC Drivers

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

- Are thread-safe
- Can implement transactions using standard JDBC statements

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

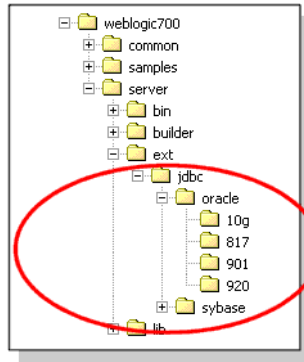
- Oracle Thin Driver 8.1.7, 9.0.1, 9.2.0, or 10g (included in WebLogic Server installation)
- Sybase jConnect Driver 4.5 and 5.5 (included in WebLogic Server installation)
- IBM Informix JDBC Driver
- Microsoft SQL Server Driver for JDBC

In WebLogic Server version 6.1, a version of the Oracle Thin Driver and the Sybase jConnect Driver were bundled within `weblogic.jar`. In version 7.x, third-party JDBC drivers are no longer bundled within `weblogic.jar`. Instead, the 10g version of the Oracle Thin driver (`classes12.zip`) and the 4.5 (`jconnect.jar`) and 5.5 (`jconn2.jar`) versions of the Sybase jConnect driver are installed in the `WL_HOME\server\lib` folder (where `WL_HOME` is the folder where WebLogic Platform is installed) with `weblogic.jar`. The manifest in `weblogic.jar` lists these files so that they are loaded when `weblogic.jar` is loaded (when the server starts).

Note: In WebLogic Server 7.0SP5, the default version of the Oracle Thin driver was changed to the 10g driver (the version in `WL_HOME\server\lib`). In WebLogic Server 7.0SP2, SP3, and SP4, the 9.2.0 version of the Oracle Thin driver was the default version of the driver. In releases of WebLogic Server 7.0 prior to the Service Pack 2 release, the 8.1.7 version of the Oracle Thin driver was the default version.

The `WL_HOME\server\ext\jdbc` folder (where `WL_HOME` is the folder where WebLogic Platform is installed) of your WebLogic Server installation includes subfolders for Oracle and Sybase JDBC drivers. See [Figure 5-1](#).

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



The `oracle` folder includes versions of the Oracle Thin driver, including the 10g version, which is also included in the `WL_HOME\server\lib` folder, as previously mentioned. You can copy one of these files to the `WL_HOME\server\lib` folder to change the version of the Oracle Thin driver or to revert to the default version. See [“Changing or Updating the Oracle Thin Driver” on page 5-5](#) for more details.

The `sybase` folder contains the 4.5 version of the Sybase jConnect driver and a subfolder with the 5.5 version of the Sybase jConnect driver and other supporting files. These drivers—`jConnect.jar` and `jconn2.jar`, without the directory structure and additional supporting files—are also included in the `WL_HOME\server\lib` folder, as previously mentioned. WebLogic Server uses the files in the `WL_HOME\server\lib` folder during runtime. You can use the additional copies in the `WL_HOME\server\ext\jdbc\sybase` folder as a backup in the event that you update the drivers with a defective or unsupported version of the driver.

If you plan to use the default version of these drivers, you do not need to make any changes. If you plan to use a different version of these drivers, you must replace the files in `WL_HOME\server\lib` with a file from `WL_HOME\server\ext\jdbc\oracle\version`, where `version` is the version of the JDBC driver you want to use, or with a file from the DBMS vendor—Oracle or Sybase.

Because the manifest in `weblogic.jar` lists the class files for the Oracle Thin driver and Sybase jConnect driver in `WL_HOME\server\lib`, the drivers are loaded when `weblogic.jar` is loaded (when the server starts). Therefore, you do not need to add

the JDBC driver to your `CLASSPATH`. If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you must add the path to the driver files to your `CLASSPATH`.

Setting the Environment for Your Third-Party JDBC Driver

If you use a third-party JDBC driver other than the Oracle Thin Driver or Sybase jConnect Driver included in the WebLogic Server 7.0 installation, you must add the path for the JDBC driver classes to your `CLASSPATH`. The following sections describe how to set your `CLASSPATH` for Windows and UNIX when using a third-party JDBC driver.

CLASSPATH for Third-Party JDBC Driver on Windows

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

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

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

CLASSPATH for Third-Party JDBC Driver on UNIX

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

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

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

Changing or Updating the Oracle Thin Driver

WebLogic Server ships with the Oracle Thin Driver version 10g (10.1.0.2.0) preconfigured and ready to use. To use a different version, you replace `WL_HOME\server\lib\classes12.zip` with a different version of the file. For example, if you want to use the 9.2.0 version of the Oracle Thin Driver, you must copy `classes12.zip` from the `WL_HOME\server\ext\jdbc\oracle\920` folder and place it in `WL_HOME\server\lib` to replace the 10g version in that folder.

Note: In WebLogic Server 7.0SP5, the default version of the Oracle Thin driver was changed to the 10g driver (the version in `WL_HOME\server\lib`). In WebLogic Server 7.0SP2, SP3, and SP4, the 9.2.0 version of the Oracle Thin driver was the default version of the driver. In releases of WebLogic Server 7.0 prior to the Service Pack 2 release, the 8.1.7 version of the Oracle Thin driver was the default version.

Follow these instructions to use Oracle Thin Driver version 9.2.0, 9.0.1, or 8.1.7:

1. In Windows Explorer or a command shell, go to the folder for the version of the driver you want to use:
 - `WL_HOME\server\ext\jdbc\oracle\920`
 - `WL_HOME\server\ext\jdbc\oracle\901`
 - `WL_HOME\server\ext\jdbc\oracle\817`
2. Copy `classes12.zip`.
3. In Windows Explorer or a command shell, go to `WL_HOME\server\lib` and replace the existing version of `classes12.zip` with the version you copied.

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

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

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

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

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

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

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

Character Set Support with `nls_charset12.zip`

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

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

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

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

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

The `nls_charset12.zip` file is installed with WebLogic Server in the `WL_HOME\server\ext\jdbc\oracle\920` and `WL_HOME\server\ext\jdbc\oracle\10g` folders (where `WL_HOME` is the folder where WebLogic Server is installed). See “[Setting the Environment for Your Third-Party JDBC Driver](#)” on page 5-4 for instructions to set your `CLASSPATH`.

Updating Sybase jConnect Driver

WebLogic Server ships with the Sybase jConnect driver versions 4.5 and 5.5 preconfigured and ready to use. To use a different version, you replace `WL_HOME\server\lib\jConnect.jar` or `jconn2.jar` with a different version of the file from the DBMS vendor.

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

- `WL_HOME\server\ext\jdbc\sybase\jConnect.jar`
- `WL_HOME\server\ext\jdbc\sybase\jConnect-5_5\classes\jconn2.jar`

Installing and Using the IBM Informix JDBC Driver

If you want to use Weblogic Server with an Informix database, BEA recommends that you use the IBM Informix JDBC driver, available from the IBM Web site at <http://www-3.ibm.com/software/data/informix/tools/jdbc/>. The IBM Informix JDBC driver is available to use for free without support. You may have to register with IBM to download the product. Download the driver from the JDBC/EMBEDDED SQLJ section, and follow the instructions in the `install.txt` file included in the downloaded zip file to install the driver.

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

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

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

5 Using Third-Party Drivers with WebLogic Server

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

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

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

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

Connection Pool Attributes when using the IBM Informix JDBC Driver

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

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

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

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

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

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

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

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

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

Programming Notes for the IBM Informix JDBC Driver

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

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

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

The Microsoft SQL Server 2000 Driver for JDBC is available for download to all licensed SQL Server 2000 customers at no charge. The driver is a Type 4 JDBC driver that supports a subset of the JDBC 2.0 Optional Package. When you install the Microsoft SQL Server 2000 Driver for JDBC, the supporting documentation is optionally installed with it. You should refer to that documentation for the most

comprehensive information about the driver. Also, see the [release manifest at `http://msdn.microsoft.com/MSDN-FILES/027/001/779/JDBCRTMReleaseManifest.htm`](http://msdn.microsoft.com/MSDN-FILES/027/001/779/JDBCRTMReleaseManifest.htm) for known issues.

Installing the MS SQL Server JDBC Driver on a Windows System

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

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

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

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

Installing the MS SQL Server JDBC Driver on a Unix System

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

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

2. Change to the temporary directory and untar the contents of the file using the following command:

```
tar -xvf mssqlserver.tar
```

3. Execute the following command to run the installation script:

```
install.ksh
```

4. Follow the instructions on the screen. When prompted to enter an installation directory, make sure you enter the full path to the directory.

5. Add the path to the following files to your CLASSPATH:

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

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

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

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

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

- Driver Name: `com.microsoft.jdbc.sqlserver.SQLServerDriver`

- URL: `jdbc:microsoft:sqlserver://server_name:1433`

- Properties:

```
user=<myuserid>  
databaseName=<dbname>  
selectMethod=cursor
```

- Password: `mypassword`

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

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

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

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

Getting a Connection with Your Third-Party Driver

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

Using Connection Pools with a Third-Party Driver

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

Creating the Connection Pool and DataSource

See [“Configuring and Using Connection Pools” on page 2-2](#) and [“Configuring and Using DataSources” on page 2-30](#) for instructions to create a JDBC connection pool and a JDBC DataSource.

Using a JNDI Lookup to Obtain the Connection

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

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

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

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

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

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

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

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

```
        catch (Exception e) {  
            // a failure occurred  
        }  
    }  
}
```

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

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see [Programming WebLogic JNDI at http://e-docs.bea.com/wls/docs70/jndi/index.html](http://e-docs.bea.com/wls/docs70/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/docs70/javadocs/weblogic/jdbc/extensions/WLConnection.html](http://e-docs.bea.com/wls/docs70/javadocs/weblogic/jdbc/extensions/WLConnection.html).

Note: BEA strongly discourages using a physical connection instead of a logical connection from a connection pool. See [“Limitations for Using a Physical Connection” on page 5-18](#).

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 5-14](#), 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.

Using Oracle Extensions with the Oracle Thin Driver

Oracle extensions provide additional proprietary methods for working with data from an Oracle database. These methods extend the standard JDBC interfaces. BEA supports the following Oracle extensions for use with the Oracle Thin driver or another driver that supports these extensions:

- `OracleStatement`
- `OracleResultSet`
- `OraclePreparedStatement`
- `OracleCallableStatement`
- `OracleArray`
- `OracleStruct`
- `OracleRef`
- `OracleBlob`
- `OracleClob`

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

Limitations When Using Oracle JDBC Extensions

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

- You can use Oracle extensions for `ARRAYs`, `REFs`, and `STRUCTs` in server-side applications that use the same JVM as the server only. You cannot use Oracle extensions for `ARRAYs`, `REFs`, and `STRUCTs` in client applications.
- You cannot create `ARRAYs`, `REFs`, and `STRUCTs` in your applications. You can only retrieve existing `ARRAY`, `REF`, and `STRUCT` objects from a database. To create these objects in your applications, you must use a non-standard Oracle descriptor object, which is not supported in WebLogic Server.

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 first 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 Extension Interfaces and Supported Methods” on page 5-36](#).

For examples showing how to access other Oracle extension methods, see the following sections:

- [“Programming with ARRAYS” on page 5-21](#)
- [“Programming with STRUCTs” on page 5-24](#)
- [“Programming with REFs” on page 5-29](#)
- [“Programming with BLOBs and CLOBs” on page 5-34](#)

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

Import Packages to Access Oracle Extensions

Import the Oracle interfaces used in this example. The `OracleConnection` and `OracleStatement` 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 javax.sql.DataSource;
import weblogic.jdbc.vendor.oracle.*;
```

Establish the Connection

Establish the database connection using JNDI, `DataSource` and connection pool objects. For information, see [“Using a JNDI Lookup to Obtain the Connection” on page 5-14](#).

```
// 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;
}
```

Programming with **ARRAYS**

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

To use ARRAYs in WebLogic Server applications:

1. Import the required classes.(See [“Import Packages to Access Oracle Extensions” on page 5-20.](#))
2. Get a connection (see [“Establish the Connection” on page 5-20](#)) and then create a statement for the connection.
3. Get the ARRAY using a result set or a callable statement.
4. Use the ARRAY as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
5. Use the standard Java methods (when used as a `java.sql.Array`) or Oracle extension methods (when cast as a `weblogic.jdbc.vendor.oracle.OracleArray`) to work with the data.

The following sections provide more details for these actions.

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

Getting an ARRAY

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

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

```
try {  
    conn = getConnection(url);  
    stmt = conn.createStatement();  
    String sql = "select * from students";  
    //Get the result set  
    rs = stmt.executeQuery(sql);  
  
    while(rs.next()) {  
        BigDecimal id = rs.getBigDecimal("student_id");  
        String name    = rs.getString("name");
```

```
        log("ArraysDAO.getStudents() -- Id = "+id.toString()+"", Student
= "+name);
//Get the array from the result set
    Array scoreArray = rs.getArray("test_scores");
    String[] scores = (String[])scoreArray.getArray();
    for (int i = 0; i < scores.length; i++) {
        log("    Test"+(i+1)+" = "+scores[i]);
    }
}
```

Updating ARRAYS in the Database

To update an ARRAY in a database, you can Follow these steps:

1. Create an array in the database using PL/SQL, if the array you want to update does not already exist in the database.
2. Get the ARRAY using a result set or a callable statement.
3. Work with the array in your Java application as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
4. Update the array in the database using the `setArray()` method for a prepared statement or a callable statement. For example:

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

Using Oracle Array Extension Methods

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

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

Programming with STRUCTs

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

Note: Please note the following limitations when using STRUCTs:

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

To use STRUCTs in WebLogic Server applications:

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

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

Getting a STRUCT

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

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

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

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

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

Using OracleStruct Extension Methods

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

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

WebLogic Server supports the following Oracle extensions:

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

Getting STRUCT Attributes

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

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

```
conn = ds.getConnection();  
stmt = conn.createStatement();  
rs = stmt.executeQuery("select * from people");  
//The third column uses an object data type.  
//Use getObject() to assign the object to an array of values.  
struct = (java.sql.Struct)(rs.getObject(2));  
Object[] attrs = ((java.sql.Struct)struct).getAttributes();  
String address = attrs[1];
```

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

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

```
java.util.Hashtable map = new java.util.Hashtable();  
map.put("NUMBER", Class.forName("java.lang.Integer"));  
map.put("VARCHAR", Class.forName("java.lang.String"));  
Object[] attrs = ((java.sql.Struct)struct).getAttributes(map);  
String address = attrs[1];
```


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

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

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

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

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

Using STRUCTs to Update Objects in the Database

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

```
conn = ds.getConnection();

stmt = conn.createStatement();

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

ps.setString (1, "Smith");

ps.setObject (2, struct);

ps.executeUpdate();
```

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

Creating Objects in the Database

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

```
conn = ds.getConnection();

stmt = conn.createStatement();
```

```
cmd = "create type ob as object (ob1 int, ob2 int)"
stmt.execute(cmd);

cmd = "create table t1 of type ob";
stmt.execute(cmd);

cmd = "insert into t1 values (5, 5)"
stmt.execute(cmd);
```

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

Automatic Buffering for STRUCT Attributes

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

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

The following example shows how to activate automatic buffering:

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

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

Programming with REFs

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

Note: Please note the following limitations when using REFs:

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

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

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

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

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

Getting a REF

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

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

```
rs.next();

//Cast as a java.sql.Ref and get REF
ref = (java.sql.Ref) rs.getRef(1);
```

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

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

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

Using OracleRef Extension Methods

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

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

WebLogic Server supports the following Oracle extensions:

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

Getting a Value

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

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

```
oracle.sql.STRUCT student1 =  
(oracle.sql.STRUCT)((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue ();  
  
Object attributes[] = student1.getAttributes();
```

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

```
java.util.Hashtable map = new java.util.Hashtable();  
  
map.put("VARCHAR", Class.forName("java.lang.String"));  
  
map.put("NUMBER", Class.forName("java.lang.Integer"));  
  
oracle.sql.STRUCT result = (oracle.sql.STRUCT)  
    ((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue (map);
```

Updating REF Values

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

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

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

```
STUDENT s1 = new STUDENT();  
  
s1.setName("Terry Green");  
  
s1.setAge(20);  
  
((weblogic.jdbc.vendor.oracle.OracleRef)ref).setValue(s1);
```

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

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

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

For example:

```
try {  
    conn = ds.getConnection();  
    stmt = conn.createStatement();  
    //Get the REF.  
    rs = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");  
    rs.next();  
    ref = (java.sql.Ref) rs.getRef(1); //cast the REF as a java.sql.Ref  
}  
  
//Create and execute the prepared statement.  
String sqlUpdate = "update t3 s2 set col = ? where s2.col.ob1=20";  
pstmt = conn.prepareStatement(sqlUpdate);  
pstmt.setRef(1, ref);  
pstmt.executeUpdate();
```

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

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

```
rs.next();

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

// Prepare the stored procedure
sql = "{call SP1 (?, ?)}";

cstmt = conn.prepareCall(sql);

// Set IN and register OUT params
cstmt.setRef(1, ref1);

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

// Execute
cstmt.execute();
```

Creating a REF in the Database

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

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

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

Programming with BLOBs and CLOBs

This section contains sample code that demonstrates how to access the OracleBlob interface. You can use the syntax of this example for the OracleBlob interface, when using methods supported by WebLogic Server. See “[Tables of Oracle Extension Interfaces and Supported Methods](#)” on page 5-36.

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](http://www.oracle.com).

Query to Select BLOB Locator from the DBMS

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

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

Declare the WebLogic Server java.sql Objects

The following code presumes the Connection is already established:

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

Begin SQL Exception Block

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

```
try {  
  
    // get our BLOB locator..  
  
    myStatement = myConnect.createStatement();  
    rs = myStatement.executeQuery(selectBlob);  
    while (rs.next()) {  
        myRegularBlob = rs.getBlob("blobCol");  
    }  
}
```



```
}

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

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

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

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

Updating a CLOB Value Using a Prepared Statement

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

Programming with Oracle Virtual Private Databases

Starting with WebLogic Server 7.0 SP3, WebLogic Server provides support for Oracle Virtual Private Databases (VPDs). A VPD is an aggregation of server-enforced, application-defined fine-grained access control, combined with a secure application context in the Oracle 9i database server.

To use VPDs in your WebLogic Server application, you would typically do the following:

1. Create a JDBC connection pool in your WebLogic Server configuration that uses either the Oracle Thin driver or the Oracle OCI driver. See [“Configuring and Administering WebLogic JDBC” on page 2-1](#) or [“Configuring JDBC Connectivity Using the Administration Console”](#) in the *Administration Guide*.

Note: If you are using an XA-enabled version of the JDBC driver, you must set `KeepXAConnTillTxComplete=true`. See [“Additional XA Connection Pool Properties”](#) in the *Administration Guide*.

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

2. Create a data source in your WebLogic Server configuration that points to the connection pool.
3. Do the following in your application:

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

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

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

/* perform application specific work */

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

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

Tables of Oracle Extension Interfaces and Supported Methods

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

Note: Typically, when a new version of the Oracle Thin driver is released, some extension methods are removed from the driver. WebLogic Server cannot support methods that are no longer included in the driver. For the Oracle 9.2.0 Thin driver (and the WebLogic Server 7.0 Service Pack 2 release), the following methods were removed:

- `OracleStatement.getAutoRollback()`
- `OracleStatement.getWaitOption()`
- `OracleConnection.isCompatibleTo816()`

Table 5-3 OracleConnection Interface

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

Table 5-3 OracleConnection Interface

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

Table 5-3 OracleConnection Interface

Extends	Method Signature
OracleConnection	void removeAllDescriptor() throws java.sql.SQLException;
extends	
java.sql.Connection	void removeDescriptor(String s) throws java.sql.SQLException;
(continued)	
	void setDefaultAutoRefetch(boolean b) throws java.sql.SQLException;
	void setDefaultExecuteBatch(int i) throws java.sql.SQLException;
	void setDefaultRowPrefetch(int i) throws java.sql.SQLException;
	void setFDO(byte[] b) throws java.sql.SQLException;
	void setIncludeSynonyms(boolean b) throws java.sql.SQLException;
	void setPhysicalStatus(boolean b) throws java.sql.SQLException;
	void setRemarksReporting(boolean b) throws java.sql.SQLException;
	void setRestrictGetTables(boolean b) throws java.sql.SQLException;
	void setStmtCacheSize(int i) throws java.sql.SQLException;
	void setStmtCacheSize(int i, boolean b) throws java.sql.SQLException;
	void setUsingXAFlag(boolean b) throws java.sql.SQLException;
	void setXAErrorFlag(boolean b) throws java.sql.SQLException;
	void shutdown(int i) throws java.sql.SQLException;
	void startup(String s, int i) throws java.sql.SQLException;

Table 5-4 OracleStatement Interface

Extends	Method Signature
OracleStatement	String getOriginalSql() throws java.sql.SQLException;
extends java.sql.Statement	String getRevisedSql() throws java.sql.SQLException; (Deprecated in Oracle 8.1.7, removed in Oracle 9i.)
	boolean getAutoRefetch() throws java.sql.SQLException;
	boolean is_value_null(boolean b, int i) throws java.sql.SQLException;
	byte getSqlKind() throws java.sql.SQLException;
	int creationState() throws java.sql.SQLException;
	int 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 5-4 OracleStatement Interface

Extends	Method Signature
OracleStatement	void defineColumnType (int i, int j, int k) throws java.sql.SQLException;
extends java.sql.Statement	void describe () throws java.sql.SQLException;
(continued)	void setAutoRefetch (boolean b) throws java.sql.SQLException;
	void setAutoRollback (int i) throws java.sql.SQLException; (Deprecated)
	void setRowPrefetch (int i) throws java.sql.SQLException;
	void setWaitOption (int i) throws java.sql.SQLException; (Deprecated)

Table 5-5 OracleResultSet Interface

Extends	Method Signature
OracleResultSet	boolean getAutoRefetch () throws java.sql.SQLException;
extends java.sql.ResultSet	int getFirstUserColumnIndex () throws java.sql.SQLException;
	void closeStatementOnClose () throws java.sql.SQLException;
	void setAutoRefetch (boolean b) throws java.sql.SQLException;
	java.sql.ResultSet getCursor (int n) throws java.sql.SQLException;
	java.sql.ResultSet getCURSOR (String s) throws java.sql.SQLException;

Table 5-6 OracleCallableStatement Interface

Extends	Method Signature
OracleCallableStatement	void clearParameters() throws java.sql.SQLException;
extends java.sql.CallableStatement	void registerIndexTableOutParameter(int i, int j, int k, int l) throws java.sql.SQLException;
	void registerOutParameter (int i, int j, int k, int l) throws java.sql.SQLException;
	java.sql.ResultSet getCursor(int i) throws java.sql.SQLException;
	java.io.InputStream getAsciiStream(int i) throws java.sql.SQLException;
	java.io.InputStream getBinaryStream(int i) throws java.sql.SQLException;
	java.io.InputStream getUnicodeStream(int i) throws java.sql.SQLException;

Table 5-7 OraclePreparedStatement Interface

Extends	Method Signature
OraclePreparedStatement extends	int getExecuteBatch() throws java.sql.SQLException;
OracleStatement and java.sql. PreparedStatement	void defineParameterType(int i, int j, int k) throws java.sql.SQLException;
	void setDisableStmtCaching(boolean b) throws java.sql.SQLException;
	void setExecuteBatch(int i) throws java.sql.SQLException;
	void setFixedCHAR(int i, String s) throws java.sql.SQLException;
	void setInternalBytes(int i, byte[] b, int j) throws java.sql.SQLException;

Table 5-8 OracleArray Interface

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

Table 5-8 OracleArray Interface

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

Table 5-9 OracleStruct Interface

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

Table 5-10 OracleRef Interface

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

Table 5-11 OracleThinBlob Interface

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

Table 5-12 OracleThinClob Interface

Extends	Method Signature
OracleThinClob extends java.sql.Clob	<pre> 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; </pre>

6 Using dbKona (Deprecated)

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 following sections describe the dbKona classes:

- [“Overview of dbKona” on page 6-1](#)
- [“The dbKona API” on page 6-4](#)
- [“Entity Relationships” on page 6-15](#)
- [“Implementing dbKona” on page 6-16](#)

Overview of dbKona

The dbKona classes provide 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

You can also use dbKona 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 Server 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, because with its objects you can 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/docs70/jdbc/index.html](http://e-docs.bea.com/wls/docs70/jdbc/index.html).

How dbKona and a JDBC Driver Interact

dbKona depends on a JDBC driver to provide and maintain a connection to a DBMS. In order to use dbKona, you must install 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/docs70/oracle/install_jdbc.html](http://e-docs.bea.com/wls/docs70/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 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 Server), 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, such as `SelectStmt` and `KeyDef`, to shield the programmer from vendor-specific SQL. By using methods in these classes, 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`. (The `DataSet` subclass `EventfulTableDataSet` is deprecated.)

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-originated).
- 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-originated).

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-originated). 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-originated index), which contain `Values` (accessible by 1-originated 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, because an insert operation does not require a `WHERE` clause. The `KeyDef` key must not contain columns that are filled or altered by the DBMS, because 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` `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 take advantage of Query-by-example functionality.
- A `TableDataSet` is populated with `Records` (accessible by 0-origin index), which contain `Values` (accessible by 1-origin 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 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 deletion 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 Server, 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-origin index). Like `Records`, `EventfulRecords` contain `Values` (accessible by a 1-origin 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 automatically 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-origin index position. Values within a record are accessible by 1-origin 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. dbKona uses the following data description objects, which are implementations of the JDBC interface:

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

Schema

When you instantiate a `DataSet`, you implicitly create the schema that describes it, and when you fetch its records, the `DataSet` 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-origin) 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 Determining:

- 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 whether 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 the `KeyDef` object
- Determining whether the `KeyDef` object has an attribute that corresponds to a particular column name or index position.

SelectStmt

You can construct a `SelectStmt` object 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

- `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 descendency 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.

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 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. Import 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. Set Properties for Making a Connection

The following code example is a method for creating the `Properties` object that is used 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 DataFormatException, 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. Make a Connection to the DBMS

You create a `Connection` object by loading the JDBC driver class with the `Class.forName()` method, and then calling the `java.sql.DriverManager.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 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. Set 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. Create 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. Fetch the Results

When you are satisfied with the data parameters, 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. Examine 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. Examine 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. Display 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. Close 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(dsl))
.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 `PreparedStatement` object is constructed with the `prepareStatement()` 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 6-31](#) 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 stmt1 = conn.createStatement();
stmt1.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;");
stmt1.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 “`imagedata`” 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, "imagedata");
    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 ints, 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();
    }
}
```

7 Testing JDBC Connections and Troubleshooting

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

- “Monitoring JDBC Connectivity” on page 7-1
- “Validating a DBMS Connection from the Command Line” on page 7-2
- “Troubleshooting JDBC” on page 7-4
- “Troubleshooting Problems with Shared Libraries on UNIX” on page 7-8
- “Using Microsoft SQL with Nested Triggers” on page 7-10

Monitoring JDBC Connectivity

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

You can also access statistics for connection pools programmatically through the `JDBCConnectionPoolRuntimeMBean`; see [WebLogic Server Partner’s Guide at `http://e-docs.bea.com/wls/docs70/isv/index.html`](http://e-docs.bea.com/wls/docs70/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 [Monitoring a WebLogic Server Domain at](#)

http://e-docs.bea.com/wls/docs70/admin_domain/monitoring.html and [Managing JDBC Connectivity at](#)
<http://e-docs.bea.com/wls/docs70/adminguide/index.html>.

For information about using MBeans, see [Programming WebLogic JMX Services at](#)
<http://e-docs.bea.com/wls/docs70/jmx/index.html>.

Validating a DBMS Connection from the Command Line

Use the `utils.dbping` BEA utility to test two-tier JDBC database connections after you install WebLogic Server. To use the `utils.dbping` utility, you must complete the installation of your JDBC driver. Make sure you have completed the following:

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

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

Syntax

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

Arguments

DBMS

Use: ORACLE or MSSQLSERVER4

user

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

password

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

DB

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

Examples

Oracle

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

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

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

If you are using `SQLNet V2`, follow this example:

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

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

Microsoft SQL Server (Type 4 driver)

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

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

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

Troubleshooting JDBC

The following sections provide troubleshooting tips.

JDBC Connections

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

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

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

Windows

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

UNIX

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

Codeset Support

WebLogic supports Oracle codesets with the following consideration:

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

For more information, see Codeset Support in [Using WebLogic JDriver for Oracle](#).

Other Problems with Oracle on UNIX

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

Thread-related Problems on UNIX

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

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

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

- If you are using Oracle drivers, use *native* threads.
- If you are using HP UNIX, upgrade to version 11.x, because there are compatibility issues with the JVM in earlier versions, such as HP UX 10.20.
- On HP UNIX, the new JDK does not append the green-threads library to the `SHLIB_PATH`. The current JDK can not find the shared library (`.sl`) unless the library is in the path defined by `SHLIB_PATH`. To check the current value of `SHLIB_PATH`, at the command line type:

```
$ echo $SHLIB_PATH
```

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

Closing JDBC Objects

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

```
try {  
  
    Driver d =  
        (Driver)Class.forName("weblogic.jdbc.oci.Driver").newInstance();  
  
    Connection conn = d.connect("jdbc:weblogic:oracle:myserver",  
                                "scott", "tiger");  
  
        Statement stmt = conn.createStatement();  
        stmt.execute("select * from emp");  
        ResultSet rs = stmt.getResultSet();  
        // do work  
  
    }  
  
    catch (Exception e) {
```



```
        // handle any exceptions as appropriate
    }

    finally {

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

    }
```

Abandoning JDBC Objects

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

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

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

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

Behavior for the second line varies depending on which service pack of WebLogic Server you are running. Before WebLogic Server 7.0SP2, the server would return a clone of the original object, which was still subject to garbage collection. After 7.0SP2, 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 Server software) on your system. This document describes problems you may encounter and suggests solutions for them.

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

WebLogic jDriver for Oracle

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

Solaris

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

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

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

HP-UX

Incorrectly Set File Permissions

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

```
% cd WL_HOME/lib/hpux11/oci817_8
% chmod 755 *.sl
```

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

```
% echo $SHLIB_PATH
```

If the directory is not listed, add it:

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

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

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

```
# cd weblogic_directory/lib/hpux11
# chatr +s enable libweblogicoci39.sl
```

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

Incorrect SHLIB_PATH

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

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

Note also that your path cannot include the path to the Oracle 8.1.7 libraries, or clashes will occur. For more instructions, see Setting Up the Environment for [Using WebLogic JDriver for Oracle](#) at

http://e-docs.bea.com/wls/docs70/oracle/install_jdbc.html.

Using Mircrosoft SQL with Nested Triggers

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

- “Exceeding the Nesting Level” on page 7-10
- “Using Triggers and EJBs” on page 7-11

For information on supported data bases and data base drivers, see [Supported Configurations](#).

Exceeding the Nesting Level

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

For example:

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

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

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

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

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

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

```
DELETE FROM CardEJBTable WHERE cardno = 1
```

Results in the following error message:

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

To work around this issue, do the following:

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

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

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

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

Using Triggers and EJBs

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

For example:

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

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

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

insert into EmployeeEJBTable values ('1',1000,'1')
insert into CardEJBTable values ('1','1')
DELETE FROM CardEJBTable WHERE cardno = 1
```

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

To work around this issue, run the following script:

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