



BEA WebLogic Server™ and WebLogic Express®

**Configuring and Using
WebLogic jDriver for
Oracle**

Copyright

Copyright © 2002 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.

Installing and Using WebLogic jDriver for Oracle

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

Contents

About This Document

Audience.....	vii
e-docs Web Site.....	viii
How to Print the Document.....	viii
Related Information.....	viii
Contact Us!	ix
Documentation Conventions	ix

1. Introduction

Overview of WebLogic jDrivers	1-1
WebLogic jDriver for Oracle	1-2
Oracle Shared Libraries.....	1-2
Distributed Transactions with the WebLogic jDriver for Oracle/XA.....	1-3

2. Configuring WebLogic jDriver for Oracle

Preparing to Use WebLogic jDriver for Oracle	2-1
Checking Software Requirements for WebLogic jDriver for Oracle	2-2
Supported Platforms.....	2-2
Requirements for Running JDBC 2.0	2-2
Checking Licensing Functionality	2-2
Setting Up the Environment for Using WebLogic jDriver for Oracle	2-3
Windows NT	2-4
Syntax.....	2-4
Example	2-4
Solaris.....	2-5
Syntax.....	2-5
Example	2-6

IBM AIX	2-6
HP-UX 11	2-7
Syntax	2-7
Example	2-8
SGI IRIX	2-9
Siemens MIPS	2-9
Compaq Tru64 UNIX	2-9
Checking Connections to the Oracle Database	2-9
Setting Up a Connection Pool	2-10
Configuring a Connection Pool with WebLogic Server Software	2-10
Using the Connection Pool in an Application	2-11
Using IDEs or Debuggers with WebLogic jDrivers	2-11
Preparing to Set Up a Development Environment and Use the WebLogic jDriver for Oracle	2-12

3. Using WebLogic jDriver for Oracle

Local Versus Distributed Transactions	3-2
Importing JDBC Packages	3-2
Setting CLASSPATH	3-3
Oracle Client Library Versions, URLs, and Driver Class Names	3-3
Connecting to an Oracle DBMS	3-4
Connecting to a Database Using WebLogic Server in a Two-Tier Configuration	3-4
Connecting Using WebLogic Server in a Multi-Tier Configuration	3-5
Connection Example	3-6
About the Connection Object	3-6
Setting Autocommit	3-7
Making a Simple SQL Query	3-7
Inserting, Updating, and Deleting Records	3-8
Creating and Using Stored Procedures and Functions	3-9
Disconnecting and Closing Objects	3-13
Working with ResultSets from Stored Procedures	3-13
Row Caching With WebLogic JDBC	3-14
Code Example	3-14
Unsupported JDBC 2.0 Methods	3-18

4. Using WebLogic jDriver for Oracle/XA in Distributed Transactions

Differences Using the WebLogic jDriver for Oracle in XA versus Non-XA Mode	4-2
Configuring JDBC XA and Non-XA Resources	4-2
JDBC/XA Resources	4-3
Non-XA JDBC Resources	4-3
Limitations of the WebLogic jDriver for Oracle XA	4-3
Implementing Distributed Transactions	4-4
Importing Packages	4-4
Finding the Data Source via JNDI	4-5
Performing a Distributed Transaction	4-5

5. Advanced Oracle Features

Allowing Mixed Case Metadata	5-2
Data Types	5-2
WebLogic Server and Oracle's NUMBER Column	5-4
Using Oracle Long Raw Data Types	5-5
Waiting on Oracle Resources	5-5
Autocommit	5-6
Transaction Isolation Levels	5-6
Codeset Support	5-7
Support for Oracle Array Fetches	5-10
Using Stored Procedures	5-10
Binding a Parameter to an Oracle Cursor	5-11
Notes on Using CallableStatement	5-13
DatabaseMetaData Methods	5-13
Support for JDBC Extended SQL	5-14
Overview of JDBC 2.0 for Oracle	5-15
Configuration Required to Support JDBC 2.0	5-16
BLOBs and CLOBs	5-16
Transaction Boundaries	5-16
BLOBs	5-17
Connection Properties	5-17
Import Statements	5-18

Initializing a BLOB Field.....	5-18
Writing Binary Data to a BLOB	5-19
Writing a BLOB Object	5-19
Reading BLOB Data	5-20
Other Methods.....	5-21
CLOBs	5-21
Codeset Support	5-21
Initializing a CLOB Field.....	5-22
Writing ASCII Data to a CLOB	5-22
Writing Unicode Data to a CLOB	5-23
Writing CLOB Objects.....	5-24
Updating a CLOB Value Using a Prepared Statement	5-24
Reading CLOB Data	5-24
Other Methods.....	5-25
Character and ASCII Streams.....	5-26
Unicode Character Streams	5-26
ASCII Character Streams	5-26
Batch Updates.....	5-27
Using Batch Updates	5-27
Clearing the Batch	5-28
Update Counts	5-28
New Date Methods	5-28

About This Document

This document describes how to install and develop applications using WebLogic jDriver for Oracle, BEA's type-2 Java Database Connectivity (JDBC) driver for the Oracle Database management system, for local and distributed transactions.

This document is organized as follows:

- [Chapter 1, "Introduction"](#)
- [Chapter 2, "Configuring WebLogic jDriver for Oracle"](#)
- [Chapter 3, "Using WebLogic jDriver for Oracle"](#)
- [Chapter 4, "Using WebLogic jDriver for Oracle/XA in Distributed Transactions"](#)
- [Chapter 5, "Advanced Oracle Features"](#)

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. It is assumed that readers are familiar with SQL, general database concepts, and Java programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the WebLogic Server Product Documentation page at <http://e-docs.bea.com/wls/docs61>.

How to Print the Document

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

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

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

Related Information

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

Contact Us!

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
<i>italics</i>	Emphasis and book titles.

Convention	Usage
monospace text	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard. <i>Examples:</i> import java.util.Enumeration; chmod u+w * config/examples/applications .java config.xml float
<i>monospace</i> <i>italic</i> text	Variables in code. <i>Example:</i> String CustomerName;
UPPERCASE TEXT	Device names, environment variables, and logical operators. <i>Examples:</i> LPT1 BEA_HOME OR
{ }	A set of choices in a syntax line.
[]	Optional items in a syntax line. <i>Example:</i> java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]
	Separates mutually exclusive choices in a syntax line. <i>Example:</i> java weblogic.deploy [list deploy undeploy update] password {application} {source}
...	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

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



1 Introduction

This document explains how to configure and use BEA's JDBC driver for the Oracle Database Management System (DBMS) with WebLogic Server. It also describes the *MultiPools* feature.

We assume you are familiar with Java, general DBMS concepts, and Structured Query Language (SQL).

This section discusses the following topics:

- [Overview of WebLogic jDrivers](#)
- [WebLogic jDriver for Oracle](#)

Overview of WebLogic jDrivers

BEA offers the following WebLogic jDrivers for use with the WebLogic Server software:

- Type 2 native JDBC driver for Oracle that includes distributed transaction capability
- Type 4 JDBC driver for Microsoft SQL Server

Type-2 drivers use client libraries supplied by a database vendor, while Type-4 drivers are pure-Java—they connect to the database server at the wire level without vendor-supplied client libraries.

You use these drivers to create the physical database connections in a connection pool. You can also use JDBC drivers from other vendors. See [Using JDBC Drivers with WebLogic Server](#) in *Programming WebLogic JDBC* at

<http://e-docs.bea.com/wls/docs81b/jdbc/intro.html#intro002>.

WebLogic jDriver for Oracle

WebLogic jDriver for Oracle, a Type 2 JDBC driver for the Oracle DBMS, is provided with the WebLogic Server software. To use this driver, you must install a complete Oracle client, including all required libraries, on the machine that will be the client to the Oracle DBMS. This Oracle client installation must contain vendor-supplied client libraries and associated files required by WebLogic Server.

Note: You must use the same version of the WebLogic jDriver for Oracle, the Oracle client, and the database management system. That is, if you use version 8.1.7 of the Oracle DBMS, you must also use the 8.1.7 version of the Oracle client and the WebLogic jDriver for Oracle.

Oracle Shared Libraries

The WebLogic Server distribution includes a choice of several BEA-supplied native libraries for WebLogic Server. Which library you choose depends on which Oracle client version is installed on your client machine and which version of the Oracle API you will use to access your Oracle server. Before you can use this driver, you must include both the BEA-supplied native library and the Oracle-supplied client libraries in your the client's `PATH` (Windows) or `shared library path` (UNIX). For more information, see [“Configuring WebLogic jDriver for Oracle” on page 2-1](#).

Distributed Transactions with the WebLogic jDriver for Oracle/XA

WebLogic Server provides a multithreaded JDBC/XA driver for Oracle Corporation's Oracle8i and Oracle9i database management systems. The WebLogic jDriver for Oracle/XA is the transaction-enabled version of WebLogic jDriver for Oracle. The WebLogic jDriver for Oracle/XA fully supports XA, the bidirectional system-level interface between a transaction manager and a resource manager of the X/Open Distributed Transaction Processing (DTP) model.

2 Configuring WebLogic jDriver for Oracle

This section discusses the following topics:

- [Preparing to Use WebLogic jDriver for Oracle](#)
- [Setting Up the Environment for Using WebLogic jDriver for Oracle](#)
- [Checking Connections to the Oracle Database](#)
- [Setting Up a Connection Pool](#)
- [Using IDEs or Debuggers with WebLogic jDrivers](#)
- [Preparing to Set Up a Development Environment and Use the WebLogic jDriver for Oracle](#)

Preparing to Use WebLogic jDriver for Oracle

Before you can use the WebLogic jDriver for Oracle, you must complete the tasks described in this section:

- [Checking Software Requirements for WebLogic jDriver for Oracle](#)
- [Checking Licensing Functionality](#)
- [Setting Up the Environment for Using WebLogic jDriver for Oracle](#)

Checking Software Requirements for WebLogic jDriver for Oracle

This section lists the software requirements in the following areas:

- [Supported Platforms](#)
- [Requirements for Running JDBC 2.0](#)

Supported Platforms

For details about the platforms, operating systems, JVMs, DBMS versions, and client libraries supported by the WebLogic jDrivers, see [WebLogic Server Certifications](http://e-docs.bea.com/wls/certifications/certifications/index.html) at <http://e-docs.bea.com/wls/certifications/certifications/index.html>.

Requirements for Running JDBC 2.0

WebLogic Server runs on an SDK that supports the JDBC 2.0 API— The JDBC 2.0 Core API and the JDBC Optional Package API that includes distributed transactions. In addition, you must use a driver that uses the Oracle Call Interface version 8 API.

Checking Licensing Functionality

To use WebLogic jDriver for Oracle, you must have the proper license. WebLogic jDriver for Oracle licensing functionality is included in the license file located in the BEA home directory where you installed this WebLogic Server. For example:

```
c:\bea\license.bea
```

If your license included WebLogic jDriver for Oracle when you installed or last updated your WebLogic Server license, no further action is required. If you are adding this functionality, you must get an updated license from your BEA sales representative. For instructions to update your license file, see [Updating Your license.bea File](#) in *Installing BEA WebLogic Server* at <http://e-docs.bea.com/wls/docs81b/install/instlic.html#instlic005>.

Note: If you use WebLogic jDriver for Oracle when WebLogic Server is not running, you must include the path to the folder where `license.bea` resides in your `CLASSPATH`.

Setting Up the Environment for Using WebLogic jDriver for Oracle

To set up your environment to support the use of WebLogic jDrivers, you must set your path variable to include pathnames for the following:

- The directory that contains the WebLogic jDriver for Oracle. (The driver file may be a native `dll`, `so`, or `sl` file, depending on your operating system.) The file containing the driver must be available to your WebLogic Server client. The name of the path variable depends on the system you are using:
 - On a Windows system, set `PATH`.
 - On most UNIX systems, set `LD_LIBRARY_PATH`.
 - On an HP-UX system, set `SHLIB_PATH`.

The directory containing the driver file varies, depending on several factors discussed in the following text.

- The directory in which vendor-supplied libraries from Oracle reside. The location of the directory containing your Oracle client libraries varies, depending on your installation. On Windows NT, the Oracle installer places these libraries in your system path.

WebLogic Server uses the `dll`, `so`, or `sl` files built with the Oracle Call Interface (OCI) version 8 API as the native interface for accessing an Oracle DBMS.

The tables in the following platform-specific sections list the directories—based on the Oracle client version—that you must specify in your system `PATH` to access the desired version of the driver.

Windows NT

Add the pathnames for the WebLogic shared library (.dll) directory and the directory where you installed the Oracle client to the PATH as follows:

Syntax

Use the following syntax:

- Add WL_HOME\server\bin\ and the appropriate WebLogic Server shared library directory from the table below to your PATH, where WL_HOME is the directory of your WebLogic Server installation. For example:

```
%WL_HOME%\server\bin\ocixxxx
```

Where xxxx is either 817_8 for Oracle 8.1.7, 901_8 for Oracle 9.0.1, or 920_8 for Oracle 9.2.0.

- Add ORACLE_HOME\bin to your PATH, where ORACLE_HOME is the directory of your Oracle client installation. Always add the WebLogic jDriver for Oracle and Oracle home information at the **beginning** of your PATH. For example:

```
%ORACLE_HOME%\bin;%PATH%
```

Example

Using the above syntax to create an actual example for Oracle 8.1.7, your path may look like:

```
$set PATH=%WL_HOME%\server\bin\oci817_8;%ORACLE_HOME%\bin;%PATH%
```

The following table provides the directory and Oracle client versions for Windows.

Table 2-1 Oracle on Windows NT

Oracle Client Version	OCI API Version	Shared Library (.dll) Directory	Notes
8.1.7	8	oci817_8	Allows access to Oracle 8 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).

Table 2-1 Oracle on Windows NT

Oracle Client Version	OCI API Version	Shared Library (.dll) Directory	Notes
9.0.1	8	oci901_8	Allows access to Oracle 9.0 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).
9.2.0	8	oci920_8	Allows access to Oracle 9.2 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).

Solaris

To set up your Solaris environment to support the use of WebLogic jDrivers, you must set your environment variable **LD_LIBRARY_PATH** to include the directory that contains the native interface file (the driver file) and the directory in which you installed the Oracle client.

Syntax

Use the following syntax:

- The directory in which the native interfaces **libweblogicoci38.so** and **libweblogicoxa38.so** reside. For example:

```
$WL_HOME/server/lib/solaris/ocixxxx
```

Where **xxxx** is either **817_8** for Oracle 8.1.7, **901_8** for Oracle 9.0.1, or **920_8** for Oracle 9.2.0.
- The directory in which vendor-supplied libraries from Oracle reside. The location of the directory containing your Oracle client libraries varies, depending on your installation. For example:

```
$ORACLE_HOME/lib
```

Example

Using the above syntax to create an actual path for Oracle 8.1.7, your path may look like:

```
export
LD_LIBRARY_PATH=$WL_HOME/server/lib/solaris/oci817_8:$ORACLE_HOME
/lib:$LD_LIBRARY_PATH
```

The following table provides the directory and Oracle client versions for Solaris.

Table 2-2 Oracle on Solaris

Oracle Client Version	OCI API Version	Shared Library (.so) Directory	Notes
8.1.7	8	oci817_8	Allows access to Oracle 8 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).
9.0.1	8	oci901_8	Allows access to Oracle 9 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).
9.2.0	8	oci920_8	Allows access to Oracle 9.2 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).

IBM AIX

To find out if your platform is supported, see [BEA WebLogic Server Certifications](http://e-docs.bea.com/wls/certifications/certifications/index.html) at <http://e-docs.bea.com/wls/certifications/certifications/index.html>.

HP-UX 11

To set up your HP environment to support the use of WebLogic jDrivers, you must set your environment variable **SHLIB_PATH** to include the directory that contains the native interface file (driver file) and the directory in which you installed your Oracle client.

Note: Oracle 9 for HP-UX is available in a 64-bit version only, including the Oracle client. Because the WebLogic jDriver for Oracle is a type-2 JDBC driver, it requires the Oracle client for database access. Therefore, to use the WebLogic jDriver for Oracle with Oracle 9, you must run WebLogic Server on a 64-bit machine.

Syntax

For Oracle 8, use the following syntax:

- The directory in which the native interfaces (driver files) **libweblogicoci38.sl** and **libweblogicoxa38.so** for Oracle 8i reside. For example:

```
$WL_HOME/server/lib/hpux11/oci817_8
```

- The directory in which vendor-supplied libraries from Oracle reside. The location of the directory containing your Oracle client libraries varies, depending on your installation. For example:

```
$ORACLE_HOME/lib
```

For Oracle 9i, use the following syntax:

- The directory in which the native interfaces (driver files) **libweblogicoci38.sl** and **libweblogicoxa38.so** for Oracle 9i reside. For example:

```
$WL_HOME/server/lib/hpux11/oci901_8
```

- The directory in which vendor-supplied libraries from Oracle reside. The location of the directory containing your Oracle client libraries varies, depending on your installation. For example:

```
$ORACLE_HOME/lib32
```

Example

Using the above syntax to create an actual path for Oracle 8.1.7, your path may look like:

```
export SHLIB_PATH=
$WL_HOME/server/lib/hpux11/oci817_8:$ORACLE_HOME/lib:$SHLIB_PATH
```

For Oracle 9.0.1, your path may look like:

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

The following table provides the directory and Oracle client versions for HP-UX.

Table 2-3 Oracle on HP

Oracle Client Version	OCI API Version	Shared Library (.sl) Directory	Notes
8.1.7	8	oci817_8	Allows access to Oracle 8 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).
9.0.1	8	oci901_8	Allows access to Oracle 9 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).
9.2.0	8	oci920_8	Allows access to Oracle 9.2 and JDBC 2.0 Core API and Optional Package API (includes distributed transactions).

SGI IRIX

To find out if your platform is supported, see [BEA WebLogic Server Certifications](http://e-docs.bea.com/wls/certifications/certifications/index.html) at <http://e-docs.bea.com/wls/certifications/certifications/index.html>.

Siemens MIPS

To find out if your platform is supported, see [BEA WebLogic Server Certifications](http://e-docs.bea.com/wls/certifications/certifications/index.html) at <http://e-docs.bea.com/wls/certifications/certifications/index.html>.

Compaq Tru64 UNIX

To find out if your platform is supported, see [BEA WebLogic Server Certifications](http://e-docs.bea.com/wls/certifications/certifications/index.html) at <http://e-docs.bea.com/wls/certifications/certifications/index.html>.

Checking Connections to the Oracle Database

Once you have installed WebLogic jDriver for Oracle, verify that you can use it to connect to your database. To test your connection, use a utility called `dbping` that is provided with the WebLogic Server software.

To set your environment and to use `dbping`, type the following commands on the command line:

```
WL_HOME\server\bin\setWLSEnv.cmd
set path=WL_HOME\server\bin\oci817_8;%PATH%
java utils.dbping ORACLE user password server
```

Where *WL_HOME* is the directory where WebLogic Platform is installed, typically `c:\bea\weblogic700`.

For detailed instructions for using the `dbping` utility, see [Using the WebLogic Java Utilities](#) in the *Command Reference Guide* at

http://e-docs.bea.com/wls/docs81b/admin_ref/utills.html.

If you have problems, check [Testing JDBC Connections and Troubleshooting](#) in *Programming WebLogic JDBC*.

Setting Up a Connection Pool

If you are using WebLogic jDriver for Oracle with either BEA WebLogic Server or BEA WebLogic Express, you can set up a pool of connections to your Oracle DBMS to be established when WebLogic Server starts. Because the connections are shared among users, these connection pools eliminate the overhead of opening a new database connection for each user.

Your application then looks up a `DataSource` on the JNDI tree and requests a connection from the connection pool. When finished with the database connection, your application returns it to the connection pool.

Configuring a Connection Pool with WebLogic Server Software

1. Include the vendor-supplied native libraries and the WebLogic native libraries for WebLogic Server in the `PATH` (Windows) or `load library path` (UNIX) of the shell where you will start WebLogic Server. For more information, see [Starting and Stopping Servers](#) in the *Administration Console Online Help* at <http://e-docs.bea.com/wls/docs81b/ConsoleHelp/startstop.html>.
2. Use the Administration Console to set up connection pools. To read about connection pools, see [JDBC Components—Connection Pools, Data Sources, and MultiPools](#) in the *Administration Console Online Help* at http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc.html#jdbc_components and [Configuring JDBC Connection Pools](#) in the *Administration*

Console Online Help at

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

Using the Connection Pool in an Application

Table 2-4

To use a connection pool in this type of application . . .	Establish a database connection using . . .	For details, see . . .
Client-side	DataSource on the JNDI tree	Configuring and Using DataSources in <i>Programming WebLogic JDBC</i> .
Server-side (such as a servlet)	WebLogic RMI, Pool, and JTS drivers, or a DataSource on the JNDI tree	Connecting To a Database Using a JDBC Connection Pool in <i>Programming WebLogic HTTP Servlets</i> .

Using IDEs or Debuggers with WebLogic jDrivers

If you are using an integrated development environment (IDE) or a debugger, copy the WebLogic-supplied native library (driver file) to a new file with a name that ends in `_g` before the file extension. For example,

- On a UNIX system, copy `libweblogicoci38.so` to `libweblogicoci38_g.so`. For distributed transactions, copy `libweblogicoxa38.so` to `libweblogicoxa38_g.so`.
- On a Windows NT platform, copy `weblogicoci38.dll` to `weblogicoci38_g.dll`. For distributed transactions, copy `weblogicoxa38.dll` to `weblogicocoxa38_g.dll`.

Preparing to Set Up a Development Environment and Use the WebLogic jDriver for Oracle

For more information, read the following:

Table 2-5

For information about . . .	See the section called . . .
Setting up a development environment for running JDBC clients	Establishing a Development Environment in <i>Developing WebLogic Server Applications</i>
Using the driver	Using WebLogic jDriver for Oracle in <i>Configuring and Using WebLogic jDriver for Oracle</i> (this guide).

3 Using WebLogic jDriver for Oracle

This section walks you through the basic tasks associated with a simple application. It also includes a code example and a list of unsupported methods:

- [Local Versus Distributed Transactions](#)
- [Importing JDBC Packages](#)
- [Setting CLASSPATH](#)
- [Oracle Client Library Versions, URLs, and Driver Class Names](#)
- [Connecting to an Oracle DBMS](#)
- [Making a Simple SQL Query](#)
- [Inserting, Updating, and Deleting Records](#)
- [Creating and Using Stored Procedures and Functions](#)
- [Disconnecting and Closing Objects](#)
- [Working with ResultSets from Stored Procedures](#)
- [Row Caching With WebLogic JDBC](#)
- [Code Example](#)
- [Unsupported JDBC 2.0 Methods](#)

Local Versus Distributed Transactions

When performing transactions with WebLogic Server, there are differences in some basic tasks, depending on whether you are using local or distributed transactions. These transactions are as follows:

- Local transactions—use the WebLogic jDriver for Oracle
- Distributed, or global, transactions—use the WebLogic jDriver for Oracle in XA mode, written as WebLogic jDriver for Oracle/XA.

For more information about distributed transactions, see [Using WebLogic jDriver for Oracle/XA in Distributed Transactions](#).

Importing JDBC Packages

The classes that you import into your application should include:

```
import java.sql.*;
import java.util.Properties; \\ required only if using a Properties
                             \\ object to set connection parameters
import weblogic.common.*;

import javax.sql.DataSource; \\required only if using DataSource
                             \\API to get connections

import javax.naming.*;       \\required only if using JNDI
                             \\to look up DataSource objects
```

The WebLogic Server driver implements the `java.sql` interface. You write your application using the `java.sql` classes. You do not need to import the JDBC driver class; instead, you load the driver inside the application. This allows you to select an appropriate driver at runtime. You can even decide what DBMS to connect to after the program is compiled.

Setting CLASSPATH

When running a WebLogic Server client using the driver provided with WebLogic Server you must put the following directory in your CLASSPATH:

```
%WL_HOME%\server\lib\weblogic.jar
```

(where %WL_HOME% is the directory where WebLogic Platform is installed, typically c:\bea\weblogic700.)

For more information about setting CLASSPATH and other issues related to setting up your environment, see [Establishing a Development Environment](#) in *Developing WebLogic Server Applications*.

Oracle Client Library Versions, URLs, and Driver Class Names

Which driver class name and URL you use depends on these factors:

- Which platform you are using
- Which version of the Oracle client libraries you are using

You must also specify the correct driver version in your system's path. For more information, see [Setting Up the Environment for Using WebLogic jDriver for Oracle](#).

When using the driver in normal (non-XA) mode:

- Driver class: `weblogic.jdbc.oci.Driver`
- URL: `jdbc:weblogic:oracle`

When using the driver in XA mode:

- Driver class: `weblogic.jdbc.oci.xa.XADataSource`
- URL: none required

Connecting to an Oracle DBMS

You make connections from your application to an Oracle DBMS using either a two-tier or multi-tier connection, as described in the following sections.

Connecting to a Database Using WebLogic Server in a Two-Tier Configuration

To make a two-tier connection from your application to an Oracle DBMS using WebLogic Server, complete the following procedure. For more information on connections, see [“Configuring a Connection Pool with WebLogic Server Software” on page 2-10](#).

1. Load the WebLogic Server **JDBC driver class**, casting it to a `java.sql.Driver` object. If you are using an XA driver, use the Datasource API, but not the `java.sql.Driver` API. For example:

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

2. Create a `java.util.Properties` object describing the connection. This object contains name-value pairs containing information such as user name, password, database name, server name, and port number. For example:

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

The server name (DEMO in the preceding example) refers to an entry in the `tnsnames.ora` file, which is located in your Oracle client installation. The server name defines host names and other information about an Oracle database. If you do not supply a server name, the system looks for an environment variable (ORACLE_SID in the case of Oracle). You may also add the server name to the URL, using the following format:

```
"jdbc:weblogic:oracle:DEMO"
```

If you specify a server with this syntax, you do not need to provide a *server* property.

You can also set properties in a single URL, for use with products such as PowerSoft's PowerJ.

3. Create a JDBC Connection object, which becomes an integral piece in your JDBC operations, by calling the `Driver.connect()` method. This method takes, as its parameters, the **URL of the driver** and the `java.util.Properties` object you created in Step 2. For example:

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

In Steps 1 and 3, you are describing the JDBC driver: in the first step, you use the full package name of the driver. Note that it is dot-delimited. In the third step, you identify the driver with its URL, which is colon-delimited. The URL must include the following string: `jdbc:weblogic:oracle`. It may also include other information, such as the server host name and the database name.

Connecting Using WebLogic Server in a Multi-Tier Configuration

To make a connection from your application to an Oracle DBMS in a WebLogic Server multi-tier configuration, complete the following procedure:

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

```
try {  
    Context ctx = new InitialContext();  
    javax.sql.DataSource ds  
        = (javax.sql.DataSource) ctx.lookup ("myDataSource");  
} catch (NamingException ex) {  
  
    // lookup failed  
}
```

2. To obtain the JDBC connection from the `DataSource` object:

```
try {  
    java.sql.Connection conn = ds.getConnection();
```

```
    } catch (SQLException ex) {  
        // obtain connection failed  
    }
```

For more information, see [Configuring and Using DataSources](#) in *Programming WebLogic JDBC*.

Connection Example

This example shows how to use a Properties object to connect to a database named myDB.

```
Properties props = new Properties();  
props.put("user", "scott");  
props.put("password", "secret");  
props.put("db", "myDB");  
  
Driver myDriver = (Driver)  
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();  
Connection conn =  
    myDriver.connect("jdbc:weblogic:oracle", props);
```

About the Connection Object

The Connection object is an important part of the application. The Connection class has constructors for many fundamental database objects that you use throughout the application; in the examples that follow, for instance, you will see the Connection object `conn` used frequently. Connecting to the database completes the initial portion of the application.

You should call the `close()` method on the Connection object as soon as you finish working with it, usually at the end of a class.

Setting Autocommit

The defaults for autocommit are described in the following table:

Table 3-1 Autocommit Defaults

Transaction Type	Autocommit Default	Change Default?	Result
Local transaction	true	yes	Changing default to false can improve performance
Distributed transaction	false	no	Do not change default. Changing default to true results in <code>SQLException</code> .

Making a Simple SQL Query

The most fundamental task in database access is to retrieve data. To retrieve data with WebLogic Server, complete the following three-step procedure:

1. Create a `Statement` to send a SQL query to the DBMS.
2. Execute the `Statement`.
3. Retrieve the results into a `ResultSet`. In this example, we execute a simple query on the `Employee` table (alias `emp`) and display data from three of the columns. We also access and display metadata about the table from which the data was retrieved. Note that we close the `Statement` at the end.

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

while (rs.next()) {
    System.out.println(rs.getString("empid") + " - " +
        rs.getString("name") + " - " +
        rs.getString("dept"));
}
```

```
ResultSetMetaData md = rs.getMetaData();

System.out.println("Number of columns: " +
    md.getColumnCount());
for (int i = 1; i <= md.getColumnCount(); i++) {
    System.out.println("Column Name: "      +
        md.getColumnName(i));
    System.out.println("Nullable: "         +
        md.isNullable(i));
    System.out.println("Precision: "        +
        md.getPrecision(i));
    System.out.println("Scale: "            +
        md.getScale(i));
    System.out.println("Size: "             +
        md.getColumnDisplaySize(i));
    System.out.println("Column Type: "      +
        md.getColumnType(i));
    System.out.println("Column Type Name: " +
        md.getColumnTypeName(i));
    System.out.println("");
}

stmt.close();
```

Inserting, Updating, and Deleting Records

We illustrate three common database tasks in this step: inserting, updating, and deleting records from a database table. We use a JDBC PreparedStatement for these operations; we create the PreparedStatement, then execute it and close it.

A PreparedStatement (subclassed from JDBC Statement) allows you to execute the same SQL over and over again with different values. PreparedStatements use the JDBC “?” syntax.

```
String inssql =
    "insert into emp(empid, name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);
for (int i = 0; i < 100; i++) {
    pstmt.setInt(1, i);
    pstmt.setString(2, "Person " + i);
    pstmt.setInt(3, i);
    pstmt.execute();
}
```

```
pstmt.close();
```

We also use a `PreparedStatement` to update records. In this example, we add the value of the counter “i” to the current value of the “dept” field.

```
String updsql =
    "update emp set dept = dept + ? where empid = ?";
PreparedStatement pstmt2 = conn.prepareStatement(updsql);
for (int i = 0; i < 100; i++) {
    pstmt2.setInt(1, i);
    pstmt2.setInt(2, i);
    pstmt2.execute();
}
pstmt2.close();
```

Finally, we use a `PreparedStatement` to delete the records that were added and then updated.

```
String delsql = "delete from emp where empid = ?";
PreparedStatement pstmt3 = conn.prepareStatement(delsql);
for (int i = 0; i < 100; i++) {
    pstmt3.setInt(1, i);
    pstmt3.execute();
}
pstmt3.close();
```

Creating and Using Stored Procedures and Functions

The type of transaction you use with WebLogic Server determines how you use stored procedures and functions:

- For local transactions—you can create, use, and drop stored procedures and functions.
- For distributed transactions (driver in XA mode)—you can execute stored procedures and functions. You cannot, however, drop and create stored procedures and functions.

First, execute a series of Statements to drop a set of stored procedures and functions from the database.

```
Statement stmt = conn.createStatement();
try {stmt.execute("drop procedure proc_squareInt");}
catch (SQLException e) {//code to handle the exception goes here;}
try {stmt.execute("drop procedure func_squareInt");}
catch (SQLException e) {//code to handle the exception goes here;}
try {stmt.execute("drop procedure proc_getresults");}
catch (SQLException e) {//code to handle the exception goes here;}
stmt.close();
```

Use a JDBC Statement to create a stored procedure or function, and then use a JDBC CallableStatement (subclass from Statement) with the JDBC “?” syntax to set IN and OUT parameters.

Note that Oracle does not natively support binding to “?” values in a SQL statement. Instead it uses “:1”, “:2”, etc. You can use either syntax in your SQL with WebLogic Server.

Stored procedure input parameters are mapped to JDBC IN parameters, using the CallableStatement.setXXX() methods, like setInt(), and the JDBC PreparedStatement “?” syntax. Stored procedure output parameters are mapped to JDBC OUT parameters, using the CallableStatement.registerOutParameter() methods and JDBC PreparedStatement “?” syntax. A parameter may be both IN and OUT, which requires both a setXXX() and a registerOutParameter() call to be done on the same parameter number.

The following example uses a JDBC Statement to create an Oracle stored procedure and then executes the stored procedure with a CallableStatement. The example uses the registerOutParameter() method to set an output parameter for the squared value.

```
Statement stmt1 = conn.createStatement();
stmt1.execute
    ("CREATE OR REPLACE PROCEDURE proc_squareInt " +
     "(field1 IN OUT INTEGER, field2 OUT INTEGER) IS " +
     "BEGIN field2 := field1 * field1; field1 := " +
     "field1 * field1; END proc_squareInt;");
stmt1.close();

// Native Oracle SQL is commented out here
// String sql = "BEGIN proc_squareInt(?, ?); END;";

// This is the correct syntax as specified by JDBC
String sql = "{call proc_squareInt(?, ?)}";
CallableStatement cstmt1 = conn.prepareCall(sql);
```

```
// Register out parameters
cstmt1.registerOutParameter(2, java.sql.Types.INTEGER);
for (int i = 0; i < 5; i++) {
    cstmt1.setInt(1, i);
    cstmt1.execute();
    System.out.println(i + " " + cstmt1.getInt(1) + " "
        + cstmt1.getInt(2));
} cstmt1.close();
```

The following example uses similar code to create and execute a stored function that squares an integer.

```
Statement stmt2 = conn.createStatement();
stmt2.execute("CREATE OR REPLACE FUNCTION func_squareInt " +
    "(field1 IN INTEGER) RETURN INTEGER IS " +
    "BEGIN return field1 * field1; " +
    "END func_squareInt;");
stmt2.close();

// Native Oracle SQL is commented out here
// sql = "BEGIN ? := func_squareInt(?); END;";

// This is the correct syntax specified by JDBC
sql = "{ ? = call func_squareInt(?)}";
CallableStatement cstmt2 = conn.prepareCall(sql);

cstmt2.registerOutParameter(1, Types.INTEGER);
for (int i = 0; i < 5; i++) {
    cstmt2.setInt(2, i);
    cstmt2.execute();
    System.out.println(i + " " + cstmt2.getInt(1) +
        " " + cstmt2.getInt(2));
}
cstmt2.close();
```

The next example uses a stored procedure named `sp_getmessages` (the code for this stored procedure is not included with this example). This stored procedure takes a message number as an input parameter, looks up the message number in a table containing the message text, and returns the message text in a `ResultSet` as an output parameter. Note that you must process all `ResultSet`s returned by a stored procedure using the `Statement.execute()` and `Statement.getResult()` methods before `OUT` parameters and return status are available.

First, set up the three parameters to the `CallableStatement`:

1. Parameter 1 (output only) is the stored procedure return value

2. Parameter 2 (input only) is the msgno argument to `sp_getmessage`
3. Parameter 3 (output only) is the message text return for the message number

```
String sql = "{ ? = call sp_getmessage(?, ?)}";
CallableStatement stmt = conn.prepareCall(sql);

stmt.registerOutParameter(1, java.sql.Types.INTEGER);
stmt.setInt(2, 18000);           // msgno 18000
stmt.registerOutParameter(3, java.sql.Types.VARCHAR);
```

Execute the stored procedure and check the return value to see if the `ResultSet` is empty. If it is not, use a loop to retrieve and display its contents.

```
boolean hasResultSet = stmt.execute();
while (true)
{
    ResultSet rs = stmt.getResultSet();
    int updateCount = stmt.getUpdateCount();
    if (rs == null && updateCount == -1) // no more results
        break;
    if (rs != null) {
        // Process the ResultSet until it is empty
        while (rs.next()) {
            System.out.println
                ("Get first col by id:" + rs.getString(1));
        }
    } else {
        // we have an update count
        System.out.println("Update count = " +
            stmt.getUpdateCount());
    }
    stmt.getMoreResults();
}
```

After you finish processing the `ResultSet`, the `OUT` parameters and return status are available.

```
int retstat = stmt.getInt(1);
String msg = stmt.getString(3);

System.out.println("sp_getmessage: status = " +
    retstat + " msg = " + msg);
stmt.close();
```


Disconnecting and Closing Objects

There are occasions when you will want to call the `commit()` method to commit changes you've made to the database before you close the connection.

When `autocommit` is set to `true` (the default JDBC transaction mode) each SQL statement is its own transaction. After we create the `Connection` for these examples, however, we set `autocommit` to `false`; in this mode, the `Connection` always has an implicit transaction associated with it, and any call to the `rollback()` or `commit()` methods will end the current transaction and start a new one. Calling `commit()` before `close()` ensures that all of the transactions are completed before closing the `Connection`.

Just as you close `Statements`, `PreparedStatement`s, and `CallableStatement`s when you have finished working with them, you should always call the `close()` method on the connection as final cleanup in your application in a `try {}` block, and you should catch exceptions and deal with them appropriately. The final two lines of this example include a call to `commit` and then a call to `close` the connection.

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

Working with ResultSets from Stored Procedures

Executing stored procedures may return multiple `ResultSets`. When you process `ResultSets` returned by a stored procedure, using `Statement.execute()` and `Statement.getResultSet()` methods, you must process all `ResultSets` returned before any of the OUT parameters or the return status codes are available.

Row Caching With WebLogic JDBC

Oracle also provides array fetching to its clients, and JDBC Driver for Oracle supports this feature. By default, JDBC Driver for Oracle will array-fetch up to 100 rows from the DBMS. This number can be altered via the property `weblogic.oci.cacheRows`.

By using the above methods, a WebLogic JDBC query for 100 rows will make only 4 calls from the client to WebLogic, and for only one of those will WebLogic actually go all the way to the DBMS for data. For more information, see [“Support for Oracle Array Fetches” on page 5-10](#).

Code Example

The following code fragments illustrate the structure for a JDBC application. The code example shown here includes retrieving data, displaying metadata, inserting, deleting, and updating data, and calling stored procedures and functions. Note the explicit calls to `close()` for each JDBC-related object, and note also that we close the `Connection` itself in a `finally {}` block, with the call to `close()` wrapped in a `try {}` block.

```
package examples.jdbc.oracle;

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

public class test {
    static int i;
    Statement stmt = null;

    public static void main(String[] argv) {
        try {
            Properties props = new Properties();
            props.put("user", "scott");
            props.put("password", "tiger");
            props.put("server", "DEMO");

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

```

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

}
catch (Exception e)
    e.printStackTrace();
}

try {
    // This will improve performance in Oracle
    // You'll need an explicit commit() call later
    conn.setAutoCommit(false);

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

    while (rs.next()) {
        System.out.println(rs.getString("empid") + " - " +
            rs.getString("name") + " - " +
            rs.getString("dept"));
    }

    ResultSetMetaData md = rs.getMetaData();

    System.out.println("Number of Columns: " +
        md.getColumnCount());
    for (i = 1; i <= md.getColumnCount(); i++) {
        System.out.println("Column Name: " +
            md.getColumnName(i));
        System.out.println("Nullable: " +
            md.isNullable(i));
        System.out.println("Precision: " +
            md.getPrecision(i));
        System.out.println("Scale: " +
            md.getScale(i));
        System.out.println("Size: " +
            md.getColumnDisplaySize(i));
        System.out.println("Column Type: " +
            md.getColumnType(i));
        System.out.println("Column Type Name: " +
            md.getColumnTypeName(i));
        System.out.println("");
    }
    rs.close();
    stmt.close();

    Statement stmtdrop = conn.createStatement();
    try {stmtdrop.execute("drop procedure proc_squareInt");}

```

```
catch (SQLException e) {}
try {stmt.drop.execute("drop procedure func_squareInt"); }
catch (SQLException e) {}
try {stmt.drop.execute("drop procedure proc_getresults"); }
catch (SQLException e) {}
stmt.drop.close();

// Create a stored procedure
Statement stmt1 = conn.createStatement();
stmt1.execute
    ("CREATE OR REPLACE PROCEDURE proc_squareInt " +
     "(field1 IN OUT INTEGER, " +
     "field2 OUT INTEGER) IS " +
     "BEGIN field2 := field1 * field1; " +
     "field1 := field1 * field1; " +
     "END proc_squareInt;");
stmt1.close();

CallableStatement cstmt1 =
    conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");
cstmt1.registerOutParameter(2, Types.INTEGER);
for (i = 0; i < 100; i++) {
    cstmt1.setInt(1, i);
    cstmt1.execute();
    System.out.println(i + " " + cstmt1.getInt(1) +
                       " " + cstmt1.getInt(2));
}
cstmt1.close();

// Create a stored function
Statement stmt2 = conn.createStatement();
stmt2.execute
    ("CREATE OR REPLACE FUNCTION func_squareInt " +
     "(field1 IN INTEGER) RETURN INTEGER IS " +
     "BEGIN return field1 * field1; END func_squareInt;");
stmt2.close();

CallableStatement cstmt2 =
    conn.prepareCall("BEGIN ? := func_squareInt(?); END;");
cstmt2.registerOutParameter(1, Types.INTEGER);
for (i = 0; i < 100; i++) {
    cstmt2.setInt(2, i);
    cstmt2.execute();
    System.out.println(i + " " + cstmt2.getInt(1) +
                       " " + cstmt2.getInt(2));
}
cstmt2.close();

// Insert 100 records
```

```

System.out.println("Inserting 100 records...");
String inssql =
    "insert into emp(empid, name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);

for (i = 0; i < 100; i++) {
    pstmt.setInt(1, i);
    pstmt.setString(2, "Person " + i);
    pstmt.setInt(3, i);
    pstmt.execute();
}
pstmt.close();

// Update 100 records
System.out.println("Updating 100 records...");
String updsql =
    "update emp set dept = dept + ? where empid = ?";
PreparedStatement pstmt2 = conn.prepareStatement(updsql);

for (i = 0; i < 100; i++) {
    pstmt2.setInt(1, i);
    pstmt2.setInt(2, i);
    pstmt2.execute();
}
pstmt2.close();

// Delete 100 records
System.out.println("Deleting 100 records...");
String delsql = "delete from emp where empid = ?";
PreparedStatement pstmt3 = conn.prepareStatement(delsql);

for (i = 0; i < 100; i++) {
    pstmt3.setInt(1, i);
    pstmt3.execute();
}
pstmt3.close();

conn.commit();
}
catch (Exception e) {
    // Deal with failures appropriately
}
finally {
    try {conn.close();}
    catch (Exception e) {
        // Catch and deal with exception
    }
}
}

```

```
}
```

For more Oracle code examples, see the `examples.jdbc.oracle` package provided with WLS in the `samples/examples` directory.

Unsupported JDBC 2.0 Methods

Although WebLogic Server supports all JDBC 2.0 methods, the WebLogic jDriver for Oracle does not support all JDBC 2.0 methods. If you need to use these methods, you can use another JDBC driver to connect to your database, such as the Oracle Thin Driver. [Table 3-2](#) lists unsupported JDBC 2.0 methods in the WebLogic jDriver for Oracle.

Table 3-2 Unsupported JDBC 2.0 Methods in the WebLogic jDriver for Oracle

Class or Interface	Unsupported Methods
java.sql.Blob	public long position(Blob blob, long l) public long position(byte abyte0[], long l)
java.sql.CallableStatement	public Array getArray(int i) public Date getDate(int i, Calendar calendar) public Object getObject(int i, Map map) public Ref getRef(int i) public Time getTime(int i, Calendar calendar) public Timestamp getTimestamp(int i, Calendar calendar) public void registerOutParameter(int i, int j, String s)
java.sql.Clob	public long position(String s, long l) public long position(java.sql.Clob clob, long l)

Table 3-2 Unsupported JDBC 2.0 Methods in the WebLogic jDriver for Oracle

Class or Interface	Unsupported Methods
java.sql.Connection	public java.sql.Statement createStatement(int i, int j) public Map getTypeMap() public CallableStatement prepareCall(String s, int i, int j) public PreparedStatement prepareStatement(String s, int i, int j) public void setTypeMap(Map map)
java.sql.DatabaseMetaData	public Connection getConnection() public ResultSet getUDTs(String s, String s1, String s2, int ai[]) public boolean supportsBatchUpdates()
java.sql.PreparedStatement	public void addBatch() public ResultSetMetaData getMetaData() public void setArray(int i, Array array) public void setNull(int i, int j, String s) public void setRef(int i, Ref ref)

Table 3-2 Unsupported JDBC 2.0 Methods in the WebLogic jDriver for Oracle

Class or Interface	Unsupported Methods
java.sql.ResultSet	public boolean absolute(int i) public void afterLast() public void beforeFirst() public void cancelRowUpdates() public void deleteRow() public boolean first() public Array getArray(int i) public Array getArray(String s) public int getConcurrency() public int getFetchDirection() public int getFetchSize() public Object getObject(int i, Map map) public Object getObject(String s, Map map) public Ref getRef(int i) public Ref getRef(String s) public int getRow() public Statement getStatement() public int getType() public void insertRow()

Table 3-2 Unsupported JDBC 2.0 Methods in the WebLogic jDriver for Oracle

Class or Interface	Unsupported Methods
java.sql.ResultSet (continued)	public boolean isAfterLast() public boolean isBeforeFirst() public boolean isFirst() public boolean isLast() public boolean last() public void moveToCurrentRow() public void moveToInsertRow() public boolean previous() public void refreshRow() public boolean relative(int i) public boolean rowDeleted() public boolean rowInserted() public boolean rowUpdated() public void setFetchDirection(int i) public void setFetchSize(int i) public void updateAsciiStream(int i, InputStream inputstream, int j) public void updateAsciiStream(String s, InputStream inputstream, int i) public void updateBigDecimal(int i, BigDecimal bigdecimal) public void updateBigDecimal(String s, BigDecimal bigdecimal) public void updateBinaryStream(int i, InputStream inputstream, int j) public void updateBinaryStream(String s, InputStream inputstream, int i) public void updateBoolean(int i, boolean flag) public void updateBoolean(String s, boolean flag) public void updateByte(int i, byte byte0) public void updateByte(String s, byte byte0) public void updateBytes(int i, byte abyte0[]) public void updateBytes(String s, byte abyte0[])

Table 3-2 Unsupported JDBC 2.0 Methods in the WebLogic jDriver for Oracle

Class or Interface	Unsupported Methods
java.sql.ResultSet (continued)	public void updateCharacterStream(int i, Reader reader, int j) public void updateCharacterStream(String s, Reader reader, int i) public void updateDate(int i, Date date) public void updateDate(String s, Date date) public void updateDouble(int i, double d) public void updateDouble(String s, double d) public void updateFloat(int i, float f) public void updateFloat(String s, float f) public void updateInt(int i, int j) public void updateInt(String s, int i) public void updateLong(int i, long l) public void updateLong(String s, long l) public void updateNull(int i) public void updateNull(String s) public void updateObject(int i, Object obj) public void updateObject(int i, Object obj, int j) public void updateObject(String s, Object obj) public void updateObject(String s, Object obj, int i) public void updateRow() public void updateShort(int i, short word0) public void updateShort(String s, short word0) public void updateString(int i, String s) public void updateString(String s, String s1) public void updateTime(int i, Time time) public void updateTime(String s, Time time) public void updateTimestamp(int i, Timestamp timestamp) public void updateTimestamp(String s, Timestamp timestamp)
java.sql.ResultSetMetaData	public String getColumnClassName(int i)

4 Using WebLogic jDriver for Oracle/XA in Distributed Transactions

The following sections describe how to integrate transactions with EJB and RMI applications that use the WebLogic jDriver for Oracle/XA and run under BEA WebLogic Server.

- [Differences Using the WebLogic jDriver for Oracle in XA versus Non-XA Mode](#)
- [Configuring JDBC XA and Non-XA Resources](#)
- [Limitations of the WebLogic jDriver for Oracle XA](#)
- [Implementing Distributed Transactions](#)

Differences Using the WebLogic jDriver for Oracle in XA versus Non-XA Mode

WebLogic jDriver for Oracle fully supports the JDBC 2.0 Optional Package API for distributed transactions. Applications using the driver in distributed transaction (XA) mode can use all JDBC 2.0 Core API the same way as in local transaction (non-XA) mode, with the exception of the following:

- Connections have to be obtained via the JDBC 2.0 `javax.sql.DataSource` API, but not through the deprecated `java.sql.DriverManager` or `java.sql.Driver` API.
- When used in WebLogic Server, you must configure a `TxDataSource` in order to use it. Refer to "[Configuring JDBC DataSources](#)" in the *Administration Console Online Help* for instructions about configuring `TxDataSource` and Connection Pools.
- Auto commit is false by default. Attempting to enable autocommit mode by calling the `java.sql.Connection.setAutoCommit` method on the Connection will throw a `SQLException`.
- Attempting to complete the distributed transaction by calling `java.sql.Connection.commit` or `java.sql.Connection.rollback` methods will throw a `SQLException`.

The reason for the last two differences is because when the WebLogic jDriver for Oracle/XA participates in a distributed transaction, it is the external Transaction Manager that is demarcating and coordinating the distributed transaction.

For more information, refer to the JDBC 2.0 Standard Extension API spec [version 1.0, dated 98/12/7 Section 7.1 last 2 paragraphs].

Configuring JDBC XA and Non-XA Resources

You use the Administration Console to configure your JDBC resources, as described in the following sections.

JDBC/XA Resources

To allow XA JDBC drivers to participate in distributed transactions, configure the JDBC connection pool as follows:

- Specify the `DriverName` property as the name of the class supporting the `javax.sql.XADataSource` interface. That is, use `weblogic.jdbc.oci.xa.XADataSource` as the `DriverName` property (`Driver Classname` in the Administration Console).
- Ensure that the database properties are specified. For more information on data source properties for the WebLogic jDriver for Oracle, see [Configuring XA JDBC Drivers for Distributed Transactions](#) in the *Administration Console Online Help* at http://e-docs.bea.com/wls/docs81b/ConsoleHelp/jdbc_connection_pools.html#jdbc007.

See the [Administration Console Online Help](#) on the JDBC Connection Pools panel for procedures and attribute definitions.

Non-XA JDBC Resources

To support non-XA JDBC resources, select the `enableTwoPhaseCommit` database property (Emulate Two-Phase Commit for non-XA Driver in the Administration Console) when configuring a JDBC Tx Data Source. For more information on this property, see [Configuring Non-XA JDBC Drivers for Distributed Transactions](#) in the *Administration Console Online Help*.

Limitations of the WebLogic jDriver for Oracle XA

WebLogic jDriver for Oracle in XA mode does not support the following:

- Mixing local and global transactions. This throws a `SQLException` if an SQL operation is attempted with no global transaction.
- Performing DDL operations (e.g. create/drop table, stored procedures, and so forth). If you want to perform DDL operations, you need to define two different connection pools as follows:
 - One non-XA connection pool that can be used for DDL operations.
 - One XA connection pool that can be used for DML operations in distributed transactions.

Implementing Distributed Transactions

This topic includes the following sections:

- [Importing Packages](#)
- [Finding the Data Source via JNDI](#)
- [Performing a Distributed Transaction](#)

Importing Packages

[Listing 4-1](#) shows the packages that the application imports. In particular, note that:

- The `java.sql.*` and `javax.sql.*` packages are required for database operations.
- The `javax.naming.*` package is required for performing a JNDI lookup on the pool name, which is passed in as a command-line parameter upon server startup. The pool name must be registered on that server group.

Listing 4-1 Importing Required Packages

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

Finding the Data Source via JNDI

[Listing 4-2](#) shows how to find the data source via JNDI.

Listing 4-2 Finding the Data Source via JNDI

```
static DataSource pool;

...

public void get_connpool(String pool_name)
    throws Exception
{
    try {
        javax.naming.Context ctx = new InitialContext();
        pool = (DataSource)ctx.lookup("jdbc/" + pool_name);
    }
    catch (javax.naming.NamingException ex){
        TP.userlog("Couldn't obtain JDBC connection pool: " +
pool_name);
        throw ex;
    }
}
```

Performing a Distributed Transaction

[Listing 4-3](#) shows a distributed transaction involving two database connections and implemented as a business method within a session bean.

Listing 4-3 Performing a Distributed Transaction

```
public class myEJB implements SessionBean {
    EJBContext ejbContext;

    public void myMethod(...) {
        javax.transaction.UserTransaction usertx;
        javax.sql.DataSource data1;
        javax.sql.DataSource data2;
        java.sql.Connection conn1;
        java.sql.Connection conn2;
        java.sql.Statement stat1;
        java.sql.Statement stat2;

        InitialContext initCtx = new InitialContext();

        //
        // Initialize a user transaction object.
        //
        usertx = ejbContext.getUserTransaction();

        //Start a new user transaction.
        usertx.begin();

        // Establish a connection with the first database
        // and prepare it for handling a transaction.
        data1 = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbc/DataBase1");
        conn1 = data1.getConnection();

        stat1 = conn1.createStatement();

        // Establish a connection with the second database
        // and prepare it for handling a transaction.
        data2 = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbc/DataBase2");
        conn2 = data2.getConnection();

        stat2 = conn2.createStatement();

        //Update both conn1 and conn2. The EJB Container
        //automatically enlists the participating resources.
        stat1.executeQuery(...);
        stat1.executeUpdate(...);
        stat2.executeQuery(...);
        stat2.executeUpdate(...);
        stat1.executeUpdate(...);
        stat2.executeUpdate(...);
    }
}
```



```
        //Commit the transaction (apply the changes to the
        //participating databases).
        usertx.commit();

        //Release all connections and statements.
        stat1.close();
        stat2.close();
        conn1.close();
        conn2.close();
    }
    ...
}
```

5 Advanced Oracle Features

This section presents advanced Oracle features for use with the WebLogic jDriver for Oracle:

- [Allowing Mixed Case Metadata](#)
- [Data Types](#)
- [WebLogic Server and Oracle's NUMBER Column](#)
- [Using Oracle Long Raw Data Types](#)
- [Waiting on Oracle Resources](#)
- [Support for JDBC Extended SQL](#)
- [Overview of JDBC 2.0 for Oracle](#)
- [Configuration Required to Support JDBC 2.0](#)
- [BLOBs and CLOBs](#)
- [Character and ASCII Streams](#)
- [New Date Methods](#)

Note: WebLogic Server also supports Oracle extension methods for prepared statements, callable statements, ARRAYs, STRUCTs, and REFs. However, to use these extensions, you must use the Oracle Thin Driver to connect to your database.

Allowing Mixed Case Metadata

WebLogic Server supports the setting of the `allowMixedCaseMetaData` property. When set to the boolean `true`, this property sets up the Connection such that mixed case is used in calls to `DatabaseMetaData` methods. If this property is set to `false`, Oracle defaults to `UPPERCASE` for database metadata.

The following sample code shows how to set up the properties to include this feature:

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

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

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

If you do not set this property, WebLogic Server defaults to the Oracle default, and `UPPERCASE` is used for database metadata.

Data Types

The following table shows the recommended mapping between Oracle data types and Java types. There are additional possibilities for representing Oracle data types in Java. If the `getObject()` method is called when result sets are being processed, it returns the default Java data type for the Oracle column being queried.

Figure 5-1 Oracle Types Mapped to WebLogic Server**Table 1:**

Oracle	WebLogic Server
Varchar	String
Number	Tinyint
Number	Smallint
Number	Integer
Number	Long
Number	Float
Number	Numeric
Number	Double
Long	Longvarchar
RowID	String
Date	Timestamp
Raw	(var)Binary
Long raw	Longvarbinary
Char	(var)Char
Boolean*	Number OR Varchar
MLS label	String
Blob	Blob
Clob	Clob

* Note that when `PreparedStatement.setBoolean()` is called, it converts a VARCHAR type to 1 or 0 (string), and it converts a NUMBER type to 1 or 0 (number).

WebLogic Server and Oracle's NUMBER Column

Oracle provides a column type called `NUMBER`, which can be optionally specified with a precision and a scale, in the forms `NUMBER(P)` and `NUMBER(P, S)`. Even in the simple unqualified `NUMBER` form, this column can hold all number types from small integer values to very large floating point numbers, with high precision.

WebLogic Server reliably converts the values in a column to the Java type requested when a WebLogic Server application asks for a value from such a column. Of course, if a value of 123.456 is asked for with `getInt()`, the value will be rounded.

The method `getObject()`, however, poses a little more complexity. WebLogic Server guarantees to return a Java object which will represent any value in a `NUMBER` column with no loss in precision. This means that a value of 1 can be returned in an `Integer`, but a value like 123434567890.123456789 can only be returned in a `BigDecimal`.

There is no metadata from Oracle to report the maximum precision of the values in the column, so WebLogic Server must decide what sort of object to return based on each value. This means that one `ResultSet` may return multiple Java types from `getObject()` for a given `NUMBER` column. A table full of integer values may all be returned as `Integer` from `getObject()`, whereas a table of floating point measurements may be returned primarily as `Double`, with some `Integer` if any value happens to be something like "123.00". Oracle does not provide any information to distinguish between a `NUMBER` value of "1" and a `NUMBER` of "1.0000000000".

There is some more reliable behavior with qualified `NUMBER` columns, that is, those defined with a specific precision. Oracle's metadata provides these parameters to the driver so WebLogic Server will always return a Java object appropriate for the given precision and scale, regardless of the values in the table.

Table 2:

Column Definition	Returned by <code>getObject()</code>
<code>NUMBER(P <= 9)</code>	<code>Integer</code>
<code>NUMBER(P <= 18)</code>	<code>Long</code>
<code>NUMBER(P = 19)</code>	<code>BigDecimal</code>

Table 2:

NUMBER(P <=16, S 0)	Double
NUMBER(P = 17, S 0)	BigDecimal

Using Oracle Long Raw Data Types

There are two properties available for use with WebLogic Server in support of Oracle's chunking of BLOBs, CLOBs, Long, and Long raw data types. Although BLOB and CLOB data types are only supported with Oracle Version 8 and JDBC 2.0, these properties also apply to Oracle's Long raw data type, which is available in Oracle Version 7.

Waiting on Oracle Resources

Note: The `waitOnResources()` method is not supported for use with the Oracle 8 API.

The WebLogic Server driver supports Oracle's `oopt()` C functionality, which allows a client to wait until resources become available. The Oracle C function sets options in cases in which requested resources are not available, such as whether to wait for locks.

A developer can specify whether a client will wait for DBMS resources, or will receive an immediate exception. The following code is an excerpt from a sample code file (`examples/jdbc/oracle/waiton.java`):

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

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

```
// You must cast the Connection as a weblogic.jdbc.oci.Connection
// to take advantage of this extension

Connection conn =(weblogic.jdbc.oci.Connection)
    myDriver.connect("jdbc:weblogic:oracle", props);

// After constructing the Connection object, immediately call
// the waitOnResources method

conn.waitOnResources(true);
```

Use of this method can cause several error return codes to be generated while the software waits for internal resources that are locked for short durations.

To take advantage of this feature, you must do the following:

1. Cast your Connection object as a `weblogic.jdbc.oci.Connection`.
2. Call the `waitOnResources()` method.

This functionality is described in section 4-97 of *The OCI Functions for C*.

Autocommit

The default transaction mode for JDBC WebLogic Server assumes autocommit to be true. You can improve the performance of your programs by setting autocommit to false, after creating a Connection object, with the following statement:

```
Connection.setAutoCommit(false);
```

Transaction Isolation Levels

WebLogic Server supports the following transaction isolation levels:

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

The Oracle DBMS supports only these two isolation levels. Unlike other JDBC drivers, WebLogic Server throws an exception if you try to use an isolation level that is unsupported. Some drivers silently ignore attempts to set an unsupported isolation level.

The `READ_UNCOMMITTED` transaction isolation level is not supported.

Codeset Support

JDBC and the WebLogic Server driver handle character strings in Java as Unicode strings. Because the Oracle DBMS uses a different codeset, the driver must convert character strings from Unicode to the codeset used by Oracle. The WebLogic Server examines the value stored in the Oracle environment variable `NLS_LANG` and select a codeset for the JDK to use for the conversion, using the mapping shown in the table called "NLS_LANG Settings Mapped to JDBC CodeSets," shown later in this section. If the `NLS_LANG` variable is not set, or if it is set to a codeset not recognized by the JDK, the driver cannot determine the correct codeset. (For information about the correct syntax for setting `NLS_LANG`, see your Oracle documentation.)

If you are converting codesets, you should pass the following property to the WebLogic Server with the `Driver.connect()` method when you establish the connection in your code:

```
props.put("weblogic.oci.min_bind_size", 660);
```

This property defines the minimum size of buffers to be bound. The default is 2000 bytes, which is also the maximum value. If you are converting codesets, you should use this property to reduce the bind size to a maximum of 660, one-third of the maximum 2000 bytes, since Oracle codeset conversion triples the buffer to allow for expansion.

WebLogic Server provides a way to set the codeset from within your Java code. By setting a connection property, `weblogic.codeset`, you can override the value stored in the `NLS_LANG` environment variable. For example, to use the `cp932` codeset, create a `Properties` object and set the `weblogic.codeset` property before calling `Driver.connect()`, as shown in the following example:

```
java.util.Properties props = new java.util.Properties();
props.put("weblogic.codeset", "cp932");
props.put("user", "scott");
props.put("password", "tiger");
```

```
String connectUrl = "jdbc:weblogic:oracle";
```

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

```
Connection conn =
    myDriver.connect(connectUrl, props);
```

Codeset support can vary with different JVMs. Check the documentation for the JDK you are using to determine whether a particular codeset is supported.

Table 5-1 NLS_LANG Settings Mapped to JDK Codesets

NLS_LANG	JDK codeset
us7ascii	ASCII
ja16sjis	SJIS
us8pc437	Cp437
we8ebcdic37	Cp1046
we8ebcdic500	Cp500
we8pc850	Cp850
we8iso8859p1	ISO8859_1
ee8iso8859p2	ISO8859_2
se8iso8859p3	ISO8859_3
nee8iso8859p4	ISO8859_4
cl8iso8859p5	ISO8859_5
ar8iso8859p6	ISO8859_6
el8iso8859p7	ISO8859_7
iw8iso8859p8	ISO8859_8
we8iso8859p9	ISO8859_9
we8iso8859p15	ISO8859_15_FDIS
ne8iso8859p10	ISO8859_10
ru8pc866	Cp866
ee8pc852	Cp852

ru8pc855	Cp855
tr8pc857	Cp857
cl8maccyrillic	MacCyrillic
we8pc860	Cp860
is8pc861	Cp861
ee8mswin1250	Cp1250
cl8mswin1251	Cp1251
el8mswin1253	Cp1253
n8pc865	Cp865
ee8macce	MacCentralEurope
ee8maccroatian	MacCroatian
tr8macturkish	MacTurkish
is8macicelandic	MacIceland
el8macgreek	MacGreek
we8macroman8	MacRoman
th8macthai	MacThai
el8pc737	Cp737
lt8pc772	Cp772
lt8pc774	Cp774
cdn8pc863	Cp863
ja16euc	EUC_JP
ko16ksc5601	EUC_KR
zht16big5	Big5
al24utffss	UTF8
utf8	UTF8

Support for Oracle Array Fetches

WebLogic Server supports Oracle array fetches. When called for the first time, `ResultSet.next()` retrieves an array of rows (rather than a single row) and stores it in memory. Each time that `next()` is called subsequently, it reads a row from the rows in memory until they are exhausted, and only then will `next()` go back to the database.

You set a property (`java.util.Property`) to control the size of the array fetch. The property is `weblogic.oci.cacheRows`; it is set by default to 100. Here's an example of setting this property to 300, which means that calls to `next()` only hit the database once for each 300 rows retrieved by the client.

```
Properties props = new Properties();
props.put("user",      "scott");
props.put("password",  "tiger");
props.put("server",    "DEMO");
props.put("weblogic.oci.cacheRows", "300");

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

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

You can improve client performance and lower the load on the database server by taking advantage of this JDBC extension. Caching rows in the client, however, requires client resources. You should tune your application for the best balance between performance and client resources, depending on your network configuration and your application.

If any columns in a `SELECT` are of type `LONG`, `BLOB`, or `CLOB`, WebLogic Server temporarily resets the cache size to 1 for the `ResultSet` associated with that select statement.

Using Stored Procedures

This section describes variations in the implementation of stored procedures that are specific to Oracle.

- Binding a Parameter to an Oracle Cursor
- Notes on Using `CallableStatement`

Binding a Parameter to an Oracle Cursor

WebLogic has created an extension to JDBC (`weblogic.jdbc.oci.CallableStatement`) that allows you to bind a parameter for a stored procedure to an Oracle cursor. You can create a JDBC `ResultSet` object with the results of the stored procedure. This allows you to return multiple `ResultSets` in an organized way. The `ResultSets` are determined at run time in the stored procedure.

Here is an example. First define the stored procedures as follows:

```
create or replace package
curs_types as
type EmpCurType is REF CURSOR RETURN emp%ROWTYPE;
end curs_types;
/

create or replace procedure
single_cursor(curs1 IN OUT curs_types.EmpCurType,
ctype in number) AS BEGIN
  if ctype = 1 then
    OPEN curs1 FOR SELECT * FROM emp;
  elsif ctype = 2 then
    OPEN curs1 FOR SELECT * FROM emp where sal < 2000;
  elsif ctype = 3 then
    OPEN curs1 FOR SELECT * FROM emp where deptno = 20;
  end if;
END single_cursor;
/

create or replace procedure
multi_cursor(curs1 IN OUT curs_types.EmpCurType,
             curs2 IN OUT curs_types.EmpCurType,
             curs3 IN OUT curs_types.EmpCurType) AS
BEGIN
  OPEN curs1 FOR SELECT * FROM emp;
  OPEN curs2 FOR SELECT * FROM emp where sal < 2000;
  OPEN curs3 FOR SELECT * FROM emp where deptno = 20;
END multi_cursor;
/
```

In your Java code, you'll construct `CallableStatements` with the stored procedures and register the output parameter as data type `java.sql.Types.OTHER`. When you retrieve the data into a `ResultSet`, use the output parameter index as an argument for the `getResultSet()` method.

```
java.sql.CallableStatement cstmt = conn.prepareCall(
    "BEGIN OPEN ? " +
    "FOR select * from emp; END;");
```

```
cstmt.registerOutParameter(1, java.sql.Types.OTHER);

cstmt.execute();
ResultSet rs = cstmt.getResultSet(1);
printResultSet(rs);
rs.close();
cstmt.close();

java.sql.CallableStatement cstmt2 = conn.prepareCall(
    "BEGIN single_cursor(?, ?); END;");
cstmt2.registerOutParameter(1, java.sql.Types.OTHER);

cstmt2.setInt(2, 1);
cstmt2.execute();
rs = cstmt2.getResultSet(1);
printResultSet(rs);

cstmt2.setInt(2, 2);
cstmt2.execute();
rs = cstmt2.getResultSet(1);}
printResultSet(rs);

cstmt2.setInt(2, 3);
cstmt2.execute();
rs = cstmt2.getResultSet(1);
printResultSet(rs);
cstmt2.close();

java.sql.CallableStatement cstmt3 = conn.prepareCall(
    "BEGIN multi_cursor(?, ?, ?); END;");
cstmt3.registerOutParameter(1, java.sql.Types.OTHER);
cstmt3.registerOutParameter(2, java.sql.Types.OTHER);
cstmt3.registerOutParameter(3, java.sql.Types.OTHER);

cstmt3.execute();

ResultSet rs1 = cstmt3.getResultSet(1);
ResultSet rs2 = cstmt3.getResultSet(2);
ResultSet rs3 = cstmt3.getResultSet(3);
```

For the full code for this example, including the `printResultSet()` method, see the examples in the `samples/examples/jdbc/oracle/` directory.

Note that the default size of an Oracle stored procedure string is 256K.

Notes on Using CallableStatement

The default length of a string bound to an OUTPUT parameter of a CallableStatement is 128 characters. If the value you assign to the bound parameter exceeds that length, you'll get the following error:

```
ORA-6502: value or numeric error
```

You can adjust the length of the value of the bound parameter by passing an explicit length with the scale argument to the

CallableStatement.registerOutputParameter() method. Here is a code example that binds a VARCHAR that will never be larger than 256 characters:

```
CallableStatement cstmt =  
    conn.prepareStatement("BEGIN testproc(?); END;");  
  
cstmt.registerOutputParameter(1, Types.VARCHAR, 256);  
cstmt.execute();  
System.out.println(cstmt.getString());  
cstmt.close();
```

DatabaseMetaData Methods

This section describes some variations in the implementation of DatabaseMetaData methods that are specific to Oracle:

- As a general rule, the String catalog argument is ignored in all DatabaseMetaData methods.
- In the DatabaseMetaData.getProcedureColumns() method:
 - The String catalog argument is ignored.
 - The String schemaPattern argument accepts only exact matches (no pattern matching).
 - The String procedureNamePattern argument accepts only exact matches (no pattern matching).
 - The String columnNamePattern argument is ignored.

Support for JDBC Extended SQL

The JavaSoft JDBC specification includes SQL Extensions, also called *SQL Escape Syntax*. All WebLogic jDrivers support Extended SQL. Extended SQL provides access to common SQL extensions in a way that is portable between DBMSs.

For example, the function to extract the day name from a date is not defined by the SQL standards. For Oracle, the SQL is:

```
select to_char(date_column, 'DAY') from table_with_dates
```

The equivalent function for Sybase and Microsoft SQL Server is:

```
select datename(dw, date_column) from table_with_dates
```

Using Extended SQL, you can retrieve the day name for both DBMSs as follows:

```
select {fn dayname(date_column)} from table_with_dates
```

Here's an example that demonstrates several features of Extended SQL:

```
String query =
"-- This SQL includes comments and " +
    "JDBC extended SQL syntax.\n" +
"select into date_table values( \n" +
"    {fn now()},           -- current time \n" +
"    {d '1997-05-24'},     -- a date      \n" +
"    {t '10:30:29' },      -- a time      \n" +
"    {ts '1997-05-24 10:30:29.123'}, -- a timestamp\n" +
"    '{string data with { or } will not be altered}'\n" +
"-- Also note that you can safely include" +
"    " { and } in comments or\n" +
"-- string data.";
Statement stmt = conn.createStatement();
stmt.executeUpdate(query);
```

Extended SQL is delimited with curly braces (“{ }”) to differentiate it from common SQL. Comments are preceded by two hyphens, and are ended by a new line (“\n”). The entire Extended SQL sequence, including comments, SQL, and Extended SQL, is placed within double quotes and passed to the `execute()` method of a `Statement` object. Here is Extended SQL used as part of a `CallableStatement`:

```
CallableStatement cstmt =
    conn.prepareCall("{ ? = call func_squareInt(?) }");
```


This example shows that you can nest extended SQL expressions:

```
select {fn dayname({fn now()})}
```

You can retrieve lists of supported Extended SQL functions from a `DatabaseMetaData` object. This example shows how to list all the functions a JDBC driver supports:

```
DatabaseMetaData md = conn.getMetaData();
System.out.println("Numeric functions:      " +
    md.getNumericFunctions());
System.out.println("\nString functions:      " +
    md.getStringFunctions());
System.out.println("\nTime/date functions: " +
    md.getTimeDateFunctions());
System.out.println("\nSystem functions:      " +
    md.getSystemFunctions());
conn.close();
```

Overview of JDBC 2.0 for Oracle

The following JDBC 2.0 features are implemented in WebLogic `JDriver` for Oracle:

- **BLOBs** (Binary Large Objects)—WebLogic Server can now handle this Oracle data type.
- **CLOBs** (Character Large Objects)—WebLogic Server can now handle this Oracle data type.
- **Character Streams** for both ASCII and Unicode characters—A better way to handle characters streams, as streams of characters instead of as byte arrays.
- **Batch Updates**—You can now send multiple statements to the database as a single unit.

These features have been added to the existing JDBC functionality previously available in the WebLogic Server. All of your existing code for previous drivers will work with the new WebLogic `JDriver` for Oracle.

Note: WebLogic Server also supports Oracle extension methods for prepared statements, callable statements, arrays, `STRUCTs`, and `REFs`. However, to use these extensions, you must use the Oracle Thin Driver to connect to your database.

Configuration Required to Support JDBC 2.0

WebLogic Server Version runs on an SDK that provides the Java 2 environment required by JDBC 2.0. For a complete list of supported configurations, see the [WebLogic Server Supported Configurations](#) page.

BLOBs and CLOBs

The BLOB (Binary Large Object) and CLOB (Character Large Object) data types were made available with the release of Oracle version 8. The JDBC 2.0 specification and WebLogic Server also support these data types. This section contains information about using these data types.

Transaction Boundaries

BLOBs and CLOBs in Oracle behave differently than other data types in regards to transactional boundaries (statements issued before an *SQL commit* or *rollback* statement). In that a BLOB or CLOB will become inactive as soon as a transaction is committed. If AutoCommit is set to TRUE, the transaction will be automatically committed after each command issued on the connection, including SELECT statements. For this reason you will need to set AutoCommit to false if you need to have a BLOB or CLOB available across multiple SQL statements. You will then need to manually commit (or rollback) the transactions at the appropriate time. To set AutoCommit to false, enter the following command:

```
conn.setAutoCommit(false); // where conn is your connection object
```

BLOBs

The BLOB data type, available with Oracle version 8, allows you to store and retrieve large binary objects in an Oracle table. Although BLOBs are defined as part of the JDBC 2.0 specification, the specification does not provide methods to update BLOB columns in a table. The BEA WebLogic implementation of BLOBs, however, does provide this functionality by means of an extension to JDBC 2.0.

Connection Properties

`weblogic.oci.selectBlobChunkSize`

This property sets the size of an internal buffer used for sending bytes or characters to an I/O stream. When the Chunk size is reached, the driver will perform an implicit `flush()` operation, which will cause the data to be sent to the DBMS.

Explicitly setting this value can be useful in controlling memory usage on the client.

If the value of this property is not explicitly set, a default value of **65534** will be used.

Set this property by passing it to the Connection object as a property. For example, this code fragment sets `weblogic.oci.selectBlobChunkSize` to 1200:

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

props.put ("weblogic.oci.selectBlobChunkSize", "1200");

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

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

`weblogic.oci.insertBlobChunkSize`

This property specifies the buffer size (in bytes) of input streams used internally by the driver.

Set this property to a positive integer to insert BLOBs into an Oracle DBMS with the BLOB chunking feature. By default, this property is set to zero (0), which means that BLOB chunking is turned off.

Import Statements

To use the BLOB functionality described in this section, import the following classes in your client code:

```
import java.sql.*;
import java.util.*;
import java.io.*;
import weblogic.jdbc.common.*;
```

Initializing a BLOB Field

When you first insert a row containing a BLOB data type, you must insert the row with an “empty” BLOB before the field can be updated with real data. You can insert an empty BLOB with the Oracle `EMPTY_BLOB()` function.

To initialize a BLOB field:

1. Create a table with one or more columns defined as a BLOB data type.
2. Insert a new row with an empty BLOB column, using the Oracle `EMPTY_BLOB()` function:

```
stmt.execute("INSERT into myTable values (1,EMPTY_BLOB());");
```

3. Obtain a “handle” to the BLOB column:

```
java.sql.Blob myBlob = null;
Statement stmt2 = conn.createStatement();
stmt2.execute("SELECT myBlobColumn from myTable");
ResultSet rs = stmt2.getResultSet();
rs.next() {
    myBlob = rs.getBlob("myBlobColumn");
    // do something with the BLOB
}
```

4. You can now write data to the BLOB. Continue with the next section, [Writing Binary Data to a BLOB](#).

Writing Binary Data to a BLOB

To write binary data to a BLOB column:

1. Obtain a handle to the BLOB field as described above, in [Initializing a BLOB Field](#), step 3.

2. Create an `InputStream` object containing the binary data.

```
java.io.InputStream is = // create your input stream
```

3. Create an output stream to which you write your BLOB data. Note that you must cast your BLOB object to `weblogic.jdbc.common.OracleBlob`.

```
java.io.OutputStream os =
((weblogic.jdbc.common.OracleBlob)
myBlob).getBinaryOutputStream();
```

4. Write the input stream containing your binary data to the output stream. The write operation is finalized when you call the `flush()` method on the `OutputStream` object.

```
byte[] inBytes = new byte[65534]; // see note below
int numBytes = is.read(inBytes);
while (numBytes > 0) {
    os.write(inBytes, 0, numBytes);
    numBytes = is.read(inBytes);
}
os.flush();
```

Note: The value [65534] in the above code presumes that you have not set the `weblogic.oci.select.BlobChunkSize` property whose default is 65534. If you have set this property, setting the `byte[]` value to match the value set in the `weblogic.oci.select.BlobChunkSize` property will provide the most efficient handling of the data. For more information about this property, see [Connection Properties](#) on page 17.

5. Clean up:

```
os.close();
pstmt.close();
conn.close();
```

Writing a BLOB Object

Writing a BLOB object to a table is performed with Prepared Statements. For example, to write the `myBlob` object to the table `myOtherTable`:

```
PreparedStatement pstmt = conn.prepareStatement(
    "UPDATE myOtherTable SET myOtherBlobColumn = ? WHERE id = 12");

pstmt.setBlob(1, myBlob);
```

Reading BLOB Data

When you retrieve a BLOB column with the `getBlob()` method and then use a `ResultSet` from a SQL `SELECT` statement, only a pointer to the BLOB data is returned; the binary data is not actually transferred to the client until the `getBinaryStream()` method is called and the data is read into the stream object.

To read BLOB data from an Oracle table:

1. Execute a `SELECT` statement:

```
stmt2.execute("SELECT myBlobColumn from myTable");
```

2. Use the results from the `SELECT` statement.

```
int STREAM_SIZE = 10;
byte[] r = new byte[STREAM_SIZE];

ResultSet rs = stmt2.getResultSet();
java.sql.Blob myBlob = null;
while (rs.next()) {
    myBlob = rs.getBlob("myBlobColumn");

    java.io.InputStream readis = myBlob.getBinaryStream();

    for (int i=0 ; i < STREAM_SIZE ; i++) {
        r[i] = (byte) readis.read();
        System.out.println("output [" + i + "] = " + r[i]);
    }
}
```

3. Clean up:

```
rs.close();
stmt2.close();
```

Note: You can also use a `CallableStatement` to generate a `ResultSet`. This `ResultSet` can then be used as shown above. See your JDK documentation under `java.sql.CallableStatement` for details.

Other Methods

The following methods of the `java.sql.Blob` interface are also implemented in the WebLogic Server JDBC 2.0 driver. For details, see your JDK documentation:

- `getBinaryStream()`
- `getBytes()`
- `length()`

The `position()` method is not implemented.

CLOBs

The CLOB data type, available with Oracle version 8, enables storage of large character strings in an Oracle table. Since the JDBC 2.0 specification does not include functionality to directly update CLOB columns, BEA has implemented the methods `getAsciiOutputStream()` (for ASCII data) and `getCharacterOutputStream()` (for Unicode data) to insert or update a CLOB.

Codeset Support

Depending on which version of the Oracle Server and client you are using you may need to set one of the following properties by passing them to the Connection object when you establish your connection the DBMS in your Java client code.

`weblogic.codeset`

This property allows you to set a codeset from within your Java code. This codeset overrides the value of any codeset specified with the Oracle environment variable `NLS_LANG`.

`weblogic.oci.ncodeset`

This property sets the National codeset used by the Oracle server. This codeset overrides the value of any codeset specified with the Oracle environment variable `NLS_NCHAR`.

`weblogic.oci.codeset_width`

This property tells the WebLogic Server which type you are using. Note the following restrictions on codeset use:

Possible Values:

- 0 for variable-width codesets
- 1 for fixed-width codesets (1 is the default value)
- 2 or 3 for the width, in bytes, of the codeset

`weblogic.oci.ncodeset_width`

If you are using one of Oracle's National codesets, specify the width of that codeset with this property. Note the following restrictions on codeset use:

Possible Values:

- 0 for variable-width codesets
- 1 for fixed-width codesets (1 is the default value)
- 2 or 3 for the width, in bytes, of the codeset

Initializing a CLOB Field

When you first insert a row containing a CLOB data type, you must insert the row with an “empty” CLOB before the field can be updated with real data. You can insert an empty CLOB with the Oracle `EMPTY_CLOB()` function.

To initialize a CLOB column:

1. Create a table with one or more columns defined as a CLOB data type.
2. Insert a new row with an empty CLOB column, using the Oracle `EMPTY_CLOB()` function:

```
stmt.execute("INSERT into myTable VALUES (1,EMPTY_CLOB())");
```

3. Obtain an object for the CLOB column:

```
java.sql.Clob myClob = null;
Statement stmt2 = conn.createStatement();
stmt2.execute("SELECT myClobColumn from myTable");
ResultSet rs = stmt2.getResultSet();
while (rs.next) {
    myClob = rs.getClob("myClobColumn");
}
```

4. You can now write character data to the CLOB. If your data is in the ASCII format, Continue with the next section, [Writing ASCII Data to a CLOB](#). If your character data is in Unicode format, see [Writing Unicode Data to a CLOB](#)

Writing ASCII Data to a CLOB

To write ASCII character data to a CLOB column:

1. Obtain a “handle” to the CLOB as described above, in [Initializing a CLOB Field](#), step 3.
2. Create an object containing the character data:


```
String s = // some ASCII data
```

3. Create an ASCII output stream to which you write your CLOB characters. Note that you must cast your CLOB object to `weblogic.jdbc.common.OracleClob`.

```
java.io.OutputStream os =  
((weblogic.jdbc.common.OracleClob)  
myclob).getAsciiOutputStream();
```

4. Write the input stream containing your ASCII data to the output stream. The write operation is finalized when you call the `flush()` method on the `OutputStream` object.

```
byte[] b = s.getBytes("ASCII");  
  
os.write(b);  
os.flush();
```

5. Clean up:

```
os.close();  
pstmt.close();  
conn.close();
```

Writing Unicode Data to a CLOB

To write Unicode character data to a CLOB column:

1. Obtain a “handle” to the CLOB as described earlier, in step 3 of “[Initializing a CLOB Field](#).”

2. Create an object containing the character data:

```
String s = // some Unicode character data
```

3. Create a character output stream to which you write your CLOB characters. Note that you must cast your CLOB object to `weblogic.jdbc.common.OracleClob`.

```
java.io.Writer wr =  
((weblogic.jdbc.common.OracleClob)  
myclob).getCharacterOutputStream();
```

4. Write the input stream containing your ASCII data to the output stream. The write operation is finalized when you call the `flush()` method on the `OutputStream` object.

```
char[] b = s.toCharArray(); // converts 's' to a character array
```

```
wr.write(b);  
wr.flush();
```

5. Clean up:

```
wr.close();  
pstmt.close();  
conn.close();
```

Writing CLOB Objects

Writing a CLOB object to a table is performed with Prepared Statements. For example, to write the `myClob` object to the table `myOtherTable`:

```
PreparedStatement pstmt = conn.prepareStatement(  
    "UPDATE myOtherTable SET myOtherClobColumn = ? WHERE id = 12");  
  
pstmt.setClob(1, myClob);
```

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.

Reading CLOB Data

When a CLOB column is retrieved using a result set from a SQL `SELECT` statement, only a pointer to the CLOB data is returned; the actual data is not transferred to the client with the result set until the `getAsciiStream()` method is called and the characters are read in to the stream.

To read CLOB data from an Oracle table:

1. Execute a `SELECT` statement:

```
java.sql.Clob myClob = null;  
Statement stmt2 = conn.createStatement();  
stmt2.execute("SELECT myClobColumn from myTable");
```

2. Use the results from the SELECT statement:

```
ResultSet rs = stmt2.getResultSet();

while (rs.next) {
    myClob = rs.getClob("myClobColumn");
    java.io.InputStream readClobis =
        myReadClob.getAsciiStream();
    char[] c = new char[26];
    for (int i=0 ; i < 26 ; i++) {
        c[i] = (char) readClobis.read();
        System.out.println("output [" + i + "] = " + c[i]);
    }
}
```

3. Clean up:

```
rs.close();
stmt2.close();
```

Note: You can also use a `CallableStatement` to generate a `ResultSet`. This `ResultSet` can then be used as shown above. See your JDK documentation under `java.sql.CallableStatment` for details.

Other Methods

The following methods of the `java.sql.Clob` interface are also implemented in the WebLogic Server (a JDBC 2.0 driver):

- `getSubString()`
- `length()`

For details about these methods, see the JDK documentation.

Note: The `position()` method is not implemented.

Character and ASCII Streams

Some new methods in the JDBC 2.0 specification allow character and ASCII streams to be manipulated as characters rather than as bytes, as in earlier versions. The following methods for handling character and ASCII streams are implemented in WebLogic Server.

Unicode Character Streams

`getCharacterStream()`

The `java.sql.ResultSet` interface uses this method for reading Unicode streams as the Java type `java.io.Reader`. This method replaces the deprecated `getUnicodeStream()` method.

`setCharacterStream()`

The `java.sql.PreparedStatement` interface uses this method for writing a `java.io.Reader` object. This method replaces the deprecated `setUnicodeStream()` method.

ASCII Character Streams

`getAsciiStream()`

The `java.sql.ResultSet` interface uses this method for reading ASCII streams as the Java type `java.io.InputStream`.

`setAsciiStream()`

The `java.sql.PreparedStatement` interface uses this method for writing a `java.io.InputStream` object.

For details about using these methods, see your JDK documentation.

Batch Updates

Batch updates are a feature of JDBC 2.0 that allows you to send multiple SQL update statements to the DBMS as a single unit. Depending on the application, this can provide improved performance over sending multiple update statements individually. The Batch update feature is available in the `Statement` interface and requires the use of SQL statements that return an update count and do *not* return a result set. Using Batch updates with the `callableStatement` or `preparedStatement` is not supported.

The following SQL statements can be used with Batch updates:

- `INSERT INTO`
- `UPDATE`
- `DELETE`
- `CREATE TABLE`
- `DROP TABLE`
- `ALTER TABLE`

Using Batch Updates

This is the basic procedure for using Batch updates:

1. Get a connection by using the WebLogic Server JDBC 2.0 driver as described in “Connecting to an Oracle DBMS” in Chapter 3, “Using WebLogic jDriver for Oracle.” For this example, the connection object is called `conn`.

2. Create a statement object using the `createStatement()` method. For example:

```
Statement stmt = conn.createStatement();
```

3. Use the `addBatch()` method to add SQL statements to the batch. These statements are not sent to the DBMS until the `executeBatch()` method is called. For example:

```
stmt.addBatch("INSERT INTO batchTest VALUES ('JOE', 20,35)");  
stmt.addBatch("INSERT INTO batchTest VALUES ('Bob', 30,44)");  
stmt.addBatch("INSERT INTO batchTest VALUES ('Ed', 34,22)");
```

4. Use the `executeBatch()` method to send the batch to the DBMS for processing. For example:

```
stmt.executeBatch();
```

If any of the statements fail an exception is thrown, and none of the statements is executed.

Clearing the Batch

You may clear a batch of statements that was created with the `addBatch()` method, by using the `clearBatch()` method. For example:

```
stmt.clearBatch();
```

Update Counts

According to the JDBC 2.0 specification, the `executeBatch()` method should return an array of Integers containing the number of rows updated for each Statement. The Oracle DBMS, however, does not supply this information to the driver. Instead, the Oracle DBMS returns `-2` for all updates.

New Date Methods

The following methods have a signature which takes a `java.util.Calendar` object as a parameter. `java.util.Calendar` allows you to specify time zone and location information that is used to translate dates. Consult your JDK API guide for details about using the `java.util.Calendar` class.

```
java.sql.ResultSet.getDate(int columnIndex, Calendar cal)  
    (returns a java.sql.Date object)
```

```
java.sql.PreparedStatement.setDate  
    (int parameterIndex, Date x, Calendar cal)
```