



# BEA WebLogic Enterprise

## Using the JDBC Drivers

WebLogic Enterprise 5.1  
Document Edition 5.1  
May 2000

## Copyright

Copyright © 2000 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, BEA Builder, BEA Jolt, BEA Manager, BEA MessageQ, BEA Tuxedo, BEA TOP END, BEA WebLogic, and ObjectBroker are registered trademarks of BEA Systems, Inc. BEA eLink, BEA eSolutions, BEA TAP, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Express, BEA WebLogic Personalization Server, BEA WebLogic Server, Java Enterprise Tuxedo, and WebLogic Enterprise Connectivity are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

## Using the JDBC Drivers

<b>Document Edition</b>	<b>Date</b>	<b>Software Version</b>
5.1	May 2000	BEA WebLogic Enterprise 5.1

---

# Contents

## About This Document

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

## 1. Using the WebLogic Enterprise JDBC/XA Drivers

Before You Begin .....	1-2
Supported Platforms .....	1-3
Adding the JAR Files to Your CLASSPATH .....	1-3
Adding Locale to Your CLASSPATH .....	1-4
Shared Libraries and Dynamic Link Libraries .....	1-4
Requirements for Making a Connection to a DBMS .....	1-5
About the Sample Code .....	1-5
About the JDBC API .....	1-6
Setting Data Source Properties .....	1-6
Administration Steps .....	1-7
Use buildXAJS to Create an XA Version of JavaServer .....	1-7
Use buildtms to Create a Transaction Manager Server Load Module for Oracle .....	1-9
Define the Database Open Information .....	1-10
Define JavaServerXA Parameters .....	1-11
Identify the Driver Class and Connection Pool Characteristics .....	1-13
Programming Steps .....	1-14
Import the Required API Packages .....	1-14

---

Initialize JavaServerXA and Get the Pool Name .....	1-15
Use a JNDI Lookup to Create a Pool of Connections.....	1-15
Get Database Connections from the Pool .....	1-16

## 2. Using JDBC Connection Pooling

About JDBC Connection Pooling.....	2-2
About the JDBC Drivers and Connection Pooling .....	2-2
UBBCONFIG Parameters for Connection Pooling .....	2-4
Sample UBBCONFIG File for Connection Pooling .....	2-5
JDBCCONNPOOLS Parameter Values .....	2-6
Encrypting DBPASSWORD and PROPS .....	2-10
Displaying Information About JDBC Connection Pools .....	2-11
T_JDBCCONNPOOLS MIB Class.....	2-13
API Characteristics .....	2-14
Application Level API.....	2-14
System Level API for the JNDI Service Provider.....	2-14
System Level API for JDBC drivers .....	2-15
Obtaining Connections from a WebLogic Enterprise Connection Pool ..	2-15
An Application's View of the Connection Lifecycle .....	2-16
The DataSource Interface .....	2-17

## 3. Using the jdbcKona Drivers

API Support .....	3-2
Platforms Supported by the jdbcKona Drivers.....	3-2
Adding the JAR Files to Your CLASSPATH .....	3-3
jdbcKona/Oracle Shared Libraries and Dynamic Link Libraries .....	3-3
Requirements for Making a Connection to a Database Management System (DBMS) .....	3-5
Support for JDBC Extended SQL .....	3-5
The JDBC API with WebLogic Extensions .....	3-6
Implementing a WebLogic Enterprise Java Application Using the jdbcKona Drivers .....	3-8
Importing Packages .....	3-9
Setting Properties for Connecting to the DBMS .....	3-9
Connecting to the DBMS .....	3-10
Making a Simple SQL Query .....	3-11

---

Inserting, Updating, and Deleting Records .....	3-12
Creating and Using Stored Procedures and Functions .....	3-13
Disconnecting and Closing Objects .....	3-15
Code Example .....	3-16

## **4. Using the jdbcKona/Oracle Drivers**

Data Type Mapping.....	4-2
Connecting a jdbcKona/Oracle Driver to an Oracle DBMS .....	4-3
Method 1 .....	4-3
Method 2 .....	4-4
Other Properties You Can Set for the jdbcKona/Oracle Driver .....	4-4
General Notes.....	4-5
Waiting for Oracle DBMS Resources .....	4-5
Autocommit.....	4-6
Using Oracle Blobs.....	4-7
Support for Oracle Array Fetches.....	4-7
Using Stored Procedures .....	4-8
Syntax for Stored Procedures in the jdbcKona/Oracle Driver .....	4-9
Binding a Parameter to an Oracle Cursor.....	4-9
Using CallableStatement .....	4-11
DatabaseMetaData Methods.....	4-12
jdbcKona/Oracle and the Oracle NUMBER Column .....	4-12

## **5. jdbcKona Extensions to the JDBC 1.22 API**

### **Index**



---

# About This Document

This document contains programming and reference information for the JDBC drivers that are provided with the BEA WebLogic Enterprise™ software.

This document covers the following topics:

- Chapter 1, “Using the WebLogic Enterprise JDBC/XA Drivers,” explains how you can use the WebLogic Enterprise JDBC/XA drivers to make local or distributed connections to Oracle 8.0.5 or 8.1.5 databases.
- Chapter 2, “Using JDBC Connection Pooling,” describes creating a pool of JDBC connections which get WebLogic Enterprise applications ready access to connections that are already open.
- Chapter 3, “Using the jdbcKona Drivers,” describes general guidelines for using the jdbcKona drivers, including a summary of the steps you take to use a jdbcKona driver in a WebLogic Enterprise Java application.
- Chapter 4, “Using the jdbcKona/Oracle Drivers,” provides guidelines for using the jdbcKona/Oracle Type 2 driver.
- Chapter 5, “jdbcKona Extensions to the JDBC 1.22 API,” describes the jdbcKona extensions to the JDBC API.

## What You Need to Know

This document is intended for programmers and system administrators who need to create and maintain transactional, scalable WebLogic Enterprise applications.

---

# e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA Systems, Inc. corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at <http://e-docs.bea.com>.

## How to Print the Document

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

A PDF version of this document is available on the WebLogic Enterprise 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 Enterprise documentation Home page, click the PDF Files button and select the document you want to print.

If you do not have Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at <http://www.adobe.com/>.

## Related Information

Before installing the BEA WebLogic Enterprise software, read the *BEA WebLogic Enterprise Release Notes*.

For more information about topics covering CORBA, Java 2 Enterprise Edition (J2EE), BEA Tuxedo®, distributed object computing, transaction processing, and Java programming, see the *WebLogic Enterprise Bibliography* in the WebLogic Enterprise online documentation.

## Contact Us!

Your feedback on the BEA WebLogic Enterprise 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 WebLogic Enterprise documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.1 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSUPPORT at **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	Item
<b>boldface text</b>	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.

---

<b>Convention</b>	<b>Item</b>
<i>italics</i>	Indicates emphasis or book titles.
monospace text	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard. <i>Examples:</i> #include <iostream.h> void main ( ) the pointer psz chmod u+w * \tux\data\ap .doc tux.doc BITMAP float
<b>monospace boldface text</b>	Identifies significant words in code. <i>Example:</i> void <b>commit</b> ( )
<i>monospace italic text</i>	Identifies variables in code. <i>Example:</i> String <i>expr</i>
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. <i>Examples:</i> LPT1 SIGNON OR
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[ ]	Indicates optional items in a syntax line. The brackets themselves should never be typed. <i>Example:</i> buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...

---

---

Convention	Item
	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
...	Indicates one of the following in a command line: <ul style="list-style-type: none"><li>■ That an argument can be repeated several times in a command line</li><li>■ That the statement omits additional optional arguments</li><li>■ That you can enter additional parameters, values, or other information</li></ul> The ellipsis itself should never be typed. <i>Example:</i> <code>buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...</code>
.	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

---



# 1 Using the WebLogic Enterprise JDBC/XA Drivers

You can use the WebLogic Enterprise JDBC/XA drivers to make local or distributed connections to Oracle 8.0.5 or 8.1.5 databases. You can use the drivers with WebLogic Enterprise CORBA Java and WebLogic Enterprise J2EE (EJB and RMI) applications.

This topic includes the following sections:

- Before You Begin
  - Supported Platforms
  - Adding the JAR Files to Your CLASSPATH
  - Adding Locale to Your CLASSPATH
  - Shared Libraries and Dynamic Link Libraries
  - Requirements for Making a Connection to a DBMS
  - About the Sample Code
  - About the JDBC API
- Setting Data Source Properties
  - Administration Steps
  - Programming Steps

For more information about JDBC connection pooling, see Chapter 2, “Using JDBC Connection Pooling.”

For more information about using transactions with the WebLogic Enterprise JDBC/XA drivers, see Transactions and the WebLogic Enterprise JDBC/XA Driver in the WebLogic Enterprise online document *Using Transactions*. It includes the following topics:

- Local versus distributed (global) transactions, with an example showing how to switch between the two types of transactions
- JDBC/XA accessibility in CORBA Methods
- JDBC/XA accessibility in EJB methods

## Before You Begin

WebLogic Enterprise applications using the WebLogic Enterprise JDBC/XA drivers can perform local transactions as well as distributed (also called global) transactions. A local transaction involves updates to a single resource manager (RM), such as a database. A distributed transaction involves updates across multiple resource managers.

Read the following topics before you start using a WebLogic Enterprise JDBC/XA driver:

- Supported Platforms
- Adding the JAR Files to Your CLASSPATH
- Shared Libraries and Dynamic Link Libraries
- Requirements for Making a Connection to a DBMS
- About the Sample Code
- About the JDBC API

---

## Supported Platforms

The WebLogic Enterprise JDBC/XA drivers for Oracle 8.0.5 and 8.1.5 are supported on the following platforms:

- Compaq Tru64 UNIX (Alpha) 4.0F
- HP-UX 11.0
- IBM AIX 4.3.3
- Microsoft Windows 2000 and NT 4.0 SP5
- Sun Solaris 2.6 and 7.0

For information about any vendor patches required on each platform, and the specific required version of the Java 2 Software Development Kit (SDK), see the WebLogic Enterprise Platform Data Sheets appendix in the WebLogic Enterprise *Installation Guide*.

## Adding the JAR Files to Your CLASSPATH

Be sure to add to your environment the WebLogic Enterprise Java ARchive (JAR) files that include the classes for the JDBC/XA drivers. You can do this by appending the following to your CLASSPATH system environment variable, where TUXDIR is the directory in which you installed the WebLogic Enterprise software:

### Windows 2000 or NT

```
%TUXDIR%\udataobj\java\jdk\M3.jar;%TUXDIR%\udataobj\java\jdk\weblogicaux.jar;
```

### UNIX

```
$TUXDIR/udataobj/java/jdk/M3.jar;$TUXDIR/udataobj/java/jdk/weblogicaux.jar;
```

## Adding Locale to Your CLASSPATH

During development, or any time you are using BEA tools, you should also set up the location for error messages from the tools. You do this by adding the following to your CLASSPATH, where TUXDIR is the directory in which you installed the WebLogic Enterprise software:

### Windows 2000 or NT

```
%TUXDIR%\locale\java\M3;
```

### UNIX

```
$TUXDIR/locale/java/M3;
```

## Shared Libraries and Dynamic Link Libraries

The JDBC/XA drivers call native libraries that are supplied with the drivers. The UNIX libraries (shared object files) are in the \$TUXDIR/lib directory. The Windows DLL files are included in the WebLogic Enterprise software kit in the %TUXDIR%\bin directory.

The following table lists the names of the driver files included with the WebLogic Enterprise Java system.

Driver Filename	Platform
weblogicoci805.dll	Microsoft Windows 2000 and NT 4.0
weblogicoci815.dll	SP5
weblogicoci805.so	Compaq Tru64 UNIX 4.0F
weblogicoci815.so	IBM AIX 4.3.3 Sun Solaris 2.6 and 7
weblogicoci805.sl	HP-UX 11.0
weblogicoci815.sl	

For the WebLogic Enterprise JDBC/XA drivers, the driver class names are `weblogic.jdbc20.oci805.Driver` or `weblogic.jdbc20.oci815.Driver`. With the JDBC 2.0 API, unlike the API for the `jdbcKona 1.22` drivers, you do not identify the driver class name in your application code. Instead, you assign data source properties, which include the driver class name. In the WebLogic Enterprise environment, this step is done by setting parameters in the WebLogic Enterprise application's `UBBCONFIG` file. For more information, see “Setting Data Source Properties” on page 1-6.

For the JDBC/XA drivers, you also need the Oracle-supplied 8.0.5 or 8.1.5 libraries for the database.

## Requirements for Making a Connection to a DBMS

You need the following components to connect to a DBMS using a JDBC/XA driver:

- An Oracle database server, version 8.0.5 or 8.1.5
- The WebLogic Enterprise JDBC/XA driver
- The Java 2 Software Developer Kit (Java 2 SDK). For information about the specific required version of the Java 2 SDK, see the WebLogic Enterprise Platform Data Sheets appendix in the WebLogic Enterprise *Installation Guide*.

## About the Sample Code

In addition to the supported sample applications that are provided with the WebLogic Enterprise software, BEA provides unsupported samples and tools on its Web site. The JDBC/XA Bankapp sample code shown in this chapter are part of the unsupported samples on the Web. For a URL pointer to the JDBC/XA Bankapp sample that is shown in this chapter, see the *BEA WebLogic Enterprise Release Notes*.

# About the JDBC API

The WebLogic Enterprise 5.1 software supports:

- The JDBC 1.22 API
- The following additional capabilities are defined in the JDBC 2.0 Optional Package API:
  - Distributed transactions: the `javax.sql.DataSource` API
  - Connection pooling
  - Java Naming and Directory Interface (JNDI)

New methods that were added in the JDBC 2.0 API, which were not present in JDBC 1.22, are not supported in this release of WebLogic Enterprise. If a WebLogic Enterprise application calls a new JDBC 2.0 method that was not in JDBC 1.22, an `SQLException` will be thrown.

# Setting Data Source Properties

The JDBC 2.0 Optional Package API, formerly known as the Standard Extension API, consists of the `javax.sql` package. This package includes the `DataSource` interface, which provides an alternative to the `DriverManager` class for making a connection to a data source. The `DriverManager` class is used with the `jdbcKona 1.22` drivers.

Using a `DataSource` implementation is better for two important reasons:

- It makes code more portable
- It makes code easier to maintain

In the WebLogic Enterprise environment, you set the data source properties separate from the WebLogic Enterprise application code. You set these properties in the application's `UBBCONFIG` configuration file. The values include the driver class name, the group in which the Java server runs, and several parameters that define the initial and run-time behavior of the JDBC connection pool.

When you create a binary `TUXCONFIG` version of the application's `UBBCONFIG` file with the `tmloadcf` command, these values are stored as `TMIB` properties. When the WebLogic Enterprise application's Java server is booted, its infrastructure will read the properties from `TMIB` and initialize the connection pools.

The setup process can be divided into:

- Administration Steps
- Programming Steps

The steps are described in subsequent sections of this topic. After you complete these steps, use the `tmloadcf` and `tmboot` commands to deploy the WebLogic Enterprise Java application, as described in *Starting and Shutting Down Applications*, an administration topic in the WebLogic Enterprise online documentation.

## Administration Steps

The administration steps are as follows:

- Use `buildXAJS` to Create an XA Version of JavaServer
- Use `buildtms` to Create a Transaction Manager Server Load Module for Oracle 8.0.5 or 8.1.5
- Define the Database Open Information
- Define JavaServerXA Parameters
- Identify the Driver Class and Connection Pool Characteristics

### Use `buildXAJS` to Create an XA Version of JavaServer

From a system prompt, use the `buildXAJS` command to build an XA resource manager that will be used with a JavaServerXA application group.

#### Syntax

```
buildXAJS [-v] -r rname [-o outfile]
```

## Example

The following example builds a JavaServerXA resource manager named `PayrollJavaServerXA` on a Solaris system:

```
prompt>buildXAJS -r Oracle_XA -o PayrollJavaServerXA
```

## Options

`-v`

Specifies that the `buildXAJS` command should work in verbose mode. In particular, it writes the build command to its standard output.

`-r rmname`

Specifies the resource manager associated with this server. If the JavaServerXA is being deployed in multithreaded mode, you must ensure that the RM contains values for Oracle 8.0.5 or 8.1.5 (matching your database software). Attempting to deploy a multithreaded JavaServerXA that is being linked with an RM other than Oracle 8.0.5 or 8.1.5 is not supported.

The value `rmname` must appear in the resource manager table located in `$TUXDIR/udataobj/RM` on UNIX systems, or `%TUXDIR%\udataobj\RM` on Windows 2000 or NT systems. On UNIX systems, each entry in this file is of the form `rmname:rmstructure_name:library_names`. On Windows 2000 or NT systems, each entry in this file is of the form `rmname;rmstructure_name;library_names`.

**Note:** See the *BEA WebLogic Enterprise Release Notes* for information about the `rmname` values that must be supplied for Oracle 8.0.5 and 8.1.5.

Using the `rmname` value, the entry in `$TUXDIR/udataobj/RM` or `%TUXDIR%\udataobj\RM` automatically includes the associated libraries for the resource manager and properly sets up the interface between the transaction manager and the resource manager.

If the `-r` option is not specified, the default is to use the null resource manager.

`-o outfile`

Specifies the name of the output file. If no name is specified, the default is `JavaServerXA`.

## Environment Variables

### TUXDIR

Finds the WebLogic Enterprise libraries and include files to use when compiling the server application.

### LD\_LIBRARY\_PATH (Solaris)

### SHLIB\_PATH (HP-UX)

### LIBPATH (IBM AIX)

Indicates which directories contain shared objects to be used by the compiler, in addition to the WebLogic Enterprise shared objects. A colon (:) is used to separate the list of directories.

### LIB (Windows 2000 or NT systems)

Indicates a list of directories within which to find libraries. A semicolon (;) is used to separate the list of directories.

## Portability

The `buildXAJLS` command is supported in UNIX and Windows 2000 or NT 4.0 systems. It is not supported on client-only WebLogic Enterprise systems.

## Use `buildtms` to Create a Transaction Manager Server Load Module for Oracle

From a system prompt, use the `buildtms` command to build a transaction manager server load module for the XA resource manager (RM). Oracle 8.0.5 or 8.1.5 is the RM that you can use with a WebLogic Enterprise JDBC/XA driver. The files that result from the `buildtms` command need to be installed in the `%TUXDIR%` directory (Windows 2000 or NT 4.0) or `$TUXDIR/bin` (UNIX) directory.

### Syntax

```
buildtms [ -v ] -o name -r rmname
```

### Examples

The following examples build transaction manager server load modules for Oracle 8.0.5 or 8.1.5.

### Windows 2000 or NT

```
prompt> buildtms -o %TUXDIR%\bin\TMS_ORA -r Oracle_XA
```

## UNIX

```
prompt> buildtms -o $TUXDIR/bin/TMS_ORA -r Oracle_XA
```

### Options

-v

Specifies that `buildtms` should work in verbose mode. In particular, it writes the `buildserver` command to its standard output and specifies the `-v` option to `buildserver`.

-o name

Specifies the filename for the output load module.

-r rmname

Specifies the resource manager associated with this server. If the JavaServerXA is being deployed in multithreaded mode, you must ensure that the RM contains values for Oracle 8.0.5 or 8.1.5. Attempting to deploy a multithreaded JavaServerXA that is being linked with an RM other than Oracle 8.0.5 or 8.1.5 is not supported.

The value `rmname` must appear in the resource manager table located in `$TUXDIR/udataobj/RM` on UNIX systems, or `%TUXDIR%\udataobj\RM` on Windows 2000 or NT systems. On UNIX systems, each entry in this file is of the form `rmname:rmstructure_name:library_names`. On Windows 2000 or NT systems, each entry in this file is of the form `rmname;rmstructure_name;library_names`.

**Note:** See the *BEA WebLogic Enterprise Release Notes* for information about the `rmname` values that must be supplied for Oracle 8.0.5 and 8.1.5.

### Portability

The `buildtms` command is supported in UNIX and Windows 2000 or NT systems. It is not supported on client-only WebLogic Enterprise systems.

## Define the Database Open Information

In the `GROUPS` section of the application's `UBBCONFIG` file, configure the `OPENINFO` parameter according to the definition of the `XA` parameter for the Oracle 8.0.5 or 8.1.5 database. Listing 1-1 shows a sample `OPENINFO` parameter value. In this example, the `OPENINFO` values are defined for a group designated as `BANK_GROUP1`.

**Listing 1-1 OPENINFO Setting in Sample UBBCONFIG**

```

*GROUPS
  SYS_GRP
    LMID      = SITE1
    GRPNO    = 1
  BANK_GROUP1
    LMID      = SITE1
    GRPNO    = 2
    OPENINFO =
"ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=..+DbgFl=0
x7+MaxCur=15+Threads=true"
    TMSNAME  = TMS_ORA
    TMSCOUNT = 2

```

In the example, note how the `TMS_ORA` name matches the Oracle transaction manager that was created for Oracle 8.0.5 or 8.1.5 in the previous `buildtms` step.

## Define JavaServerXA Parameters

In the `SERVERS` section of the application's `UBBCONFIG` file, define parameters to indicate how this WebLogic Enterprise application will use JavaServerXA.

The JavaServerXA parameter is required if you are using the JDBC/XA driver. If you want the JavaServerXA to be multithreaded, specify the `-M` option for the `CLOPT` parameter. To deploy a single-threaded JavaServerXA server, do not use the `-M` option.

Each JavaServerXA can only host WebLogic Enterprise JDBC connection pools that connect to one resource manager. The current release supports the Oracle 8.0.5 or 8.1.5 XA resource manager only.

You must specify `SRVTYPE=JAVA` for the JavaServer or JavaServerXA to use JDBC connection pooling.

**Note:** In prior WebLogic Enterprise releases, you could also specify the name of the connection pool on the command-line options (`CLOPT`) of the JavaServerXA or JavaServer parameter. Although this is still allowed by the syntax parser, the new syntax is to list the connection pool's name and the application's JAR filename in the `MODULES` section.

Listing 1-2 shows an example of JavaServerXA configured for multithreading in a sample `UBBCONFIG`. Notice the association between the `BANK_GROUP1` group that was defined in the previous listing; it contains the `OPENINFO` values.

# 1 Using the WebLogic Enterprise JDBC/XA Drivers

---

Also notice how in the `MODULES` section, the `bank_pool` value is listed to give a handle for a JDBC connection pool.

## Listing 1-2 Multithreaded Server Configuration in Sample UBBCONFIG

---

```
*SERVERS
  DEFAULT:
    RESTART = Y
    MAXGEN  = 5
    .
    .
    .
  JavaServerXA
    SRVGRP = BANK_GROUP1
    SRVID  = 2
    SRVTYPE = JAVA
    CLOPT  = "-A -- -M 10"
    RESTART = N
    .
    .
    .
*MODULES
  BankApp
    SRVGRP = BANK_GROUP1
    SRVID  = 2
    FILE   = "BankApp.jar"
    ARGS  = "TellerFactory_1 bank_pool"
```

The `-M 10` parameter for the `JavaServerXA` enables a multithreaded `JavaServer` with a pool of 10 threads. The threads setting is in addition to the JDBC connection pool settings, which are defined in the next section, `JDBCCONNPOOLS`.

**Note:** The potential for a performance gain from a multithreaded `JavaServer` depends on the application pattern and whether the application is running on a single-processor or multiprocessor machine. For more information about enabling multithreaded `JavaServers`, see *Creating a Configuration File*, an administration topic in the WebLogic Enterprise online documentation.

## Identify the Driver Class and Connection Pool Characteristics

The `JDBCONNPOOLS` section of the application's `UBBCONFIG` file includes several parameters to set the properties of the JDBC connection pool and the driver it uses. This `JDBCONNPOOLS` section was added to the WebLogic Enterprise product in version 5.1. Listing 1-3 shows a sample section.

### Listing 1-3 Defining Driver and Connection Pooling Properties in a Sample `UBBCONFIG`

```
*JDBCONNPOOLS
  bank_pool
    SRVGRP           = BANK_GROUP1
    SRVID            = 2
    DRIVER           = "weblogic.jdbc20.oci815.Driver"
    URL              = "jdbc:weblogic:oracle:beq-local"
    PROPS            = "user=scott;password=tiger;server=Beq-Local"
    ENABLEXA         = Y
    INITCAPACITY     = 2
    MAXCAPACITY      = 10
    CAPACITYINCR     = 1
    CREATEONSTARTUP = Y
```

The `SRVGRP` parameter value, `BANK_GROUP1`, forms the association between this `bank_pool` connection pool and the definition of `BANK_GROUP1` in the `GROUPS` section of the `UBBCONFIG`. The definition for `BANK_GROUP1` included the Oracle database `OPENINFO` values.

The JDBC connection pool is named `bank_pool`, which was also identified in the `MODULES` section. The use of the server ID (`SRVID = 2`) and the `bank_pool` name forms the association between the `JavaServerXA` that will run in `BANK_GROUP1` and this connection pool.

For the WebLogic Enterprise JDBC/XA driver, set the `DRIVER` parameter value to either `weblogic.jdbc20.oci805.Driver` or `weblogic.jdbc20.oci815.Driver`. Also, set the `ENABLEXA` parameter to `Y`.

See Chapter 2, "Using JDBC Connection Pooling" for information about the other `JDBCONNPOOLS` parameters. They include settings for the initial and run-time behavior of the named connection pool.

## Programming Steps

The programming steps include the following:

- Import the required API packages
- Initialize JavaServerXA and Get the Name of the JDBC Connection Pool
- Use a JNDI lookup to create a pool of connections
- Get database connection from the pool

After you complete these steps described in these sections, use the `tmloadcf` and `tmboot` commands to deploy the WebLogic Enterprise Java application, as described in *Starting and Shutting Down Applications*, an administration topic in the WebLogic Enterprise online documentation. The steps include:

- Using `tmloadcf` to create a binary version of the `UBBCONFIG` file
- Using `tmboot -y` to boot the application

## Import the Required API Packages

Listing 1-4 shows the packages that a Java 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 1-4 Importing Required Packages

---

```
import java.io.*;
import java.net.URL;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import com.beasys.*;
import com.beasys.Tobj.*;
import com.beasys.Tobj.TP;
```

## Initialize JavaServerXA and Get the Pool Name

In your Java application code, initialize the JavaServerXA. In the sample file `BankAppServerImpl.java`, when the sample `BankApp` JavaServerXA is initialized, it:

- Creates a Teller factory object reference
- Register the factory reference with the `FactoryFinder`
- Establishes connections to the database

The two arguments are:

- `TellerFactory_1`
- `bank_pool`

The `JdbcConnPoolName` argument is the name of the connection pool that was specified in the application's `UBBCONFIG` file. The JavaServerXA returns this value to the program.

For example, the following code fragment is from the sample file `BankAppServerImpl.java`:

```
public boolean initialize(String[] args)
{
    try {
        // get input arguments
        if(args.length < 2 ) {
            TP.userlog("Not enough arguments");
            TP.userlog("Correct Argument list: ");
            TP.userlog("TellerFactory_1 bank_pool");
            return false;
        }

        tellerFName = new String(args[0]);
        String pool_name = args[1];

        // write the input arguments to ULOG file
        TP.userlog("Input Arguments for Server.initialize(): ");
        TP.userlog("TellerFactory_1: " + tellerFactory_1 );
        TP.userlog("JDBC connection pool name: " + bank_pool);
    }
}
```

## Use a JNDI Lookup to Create a Pool of Connections

The sample program `BankAppServerImpl.java` uses the static variable `DataSource`, the connection pool object. For example:

```
static DataSource pool;
```

Using `DataSource`, the sample uses a JNDI lookup to create a pool of connections to the database. For example:

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

## Get Database Connections from the Pool

In the `DataSource` implementation, the `Connection` object that is returned by the `DataSource.getConnection` method is identical to a `Connection` object returned by the `jdbcKona 1.22 DriverManager.getConnection` method. Because of the advantages it offers, using a `DataSource` object is the recommended way to obtain a connection.

For the application programmer, using a `DataSource` object is a matter of choice. However, programmers writing WebLogic Enterprise JDBC applications that include distributed (XA) transactions and connection pooling must use a `DataSource` object to get connections.

For example, the following code fragment is from the sample application file `DBAccessImpl.java`:

```
public void get_valid_accounts(short pinNo, CustAccountsHolder
accounts)
    throws DataBaseException, PinNumberNotFound
{
    Statement stmt=null;
    ResultSet rs=null;
    Connection con= null;

    try {
```

```
con = BankAppServerImpl.pool.getConnection();
// Construct and execute the SQL SELECT statement.
stmt = con.createStatement();
String stmtBuf =
    new String(
        "SELECT CheckingAccountID, SavingsAccountID "
        + "FROM Cust_Data WHERE PinNo = "
        + pinNo);
rs = stmt.executeQuery(stmtBuf);
```



# 2 Using JDBC Connection Pooling

This topic includes the following sections:

- About JDBC Connection Pooling
- About the JDBC Drivers and Connection Pooling
- UBBCONFIG Parameters for Connection Pooling
  - Sample UBBCONFIG File for Connection Pooling
  - JDBCCONNPOOLS Parameter Values
  - Encrypting DBPASSWORD and PROPS
- Displaying Information About JDBC Connection Pools
- T\_JDBCCONNPOOLS MIB Class
- API Characteristics

Chapter 1, “Using the WebLogic Enterprise JDBC/XA Drivers,” describes the JDBC/XA drivers provided with the WebLogic Enterprise software and introduces the use of connection pooling with those XA drivers for Oracle 8.0.5 and 8.1.5. This chapter explains how to use connection pooling with any JDBC driver supported by the WebLogic Enterprise software. This includes the WebLogic Enterprise JDBC/XA drivers and the jdbcKona drivers that are documented in Chapter 4, “Using the jdbcKona/Oracle Drivers.”

If you use the jdbcKona drivers, it is not mandatory that you use JDBC connection pooling to obtain database connections. However, BEA recommends that you use the connection pooling feature with Java applications. If you use a JDBC/XA driver, you must use the JDBC connection pooling to obtain database connections.

## About JDBC Connection Pooling

To conserve system resources and to improve the performance of transactional BEA WebLogic Enterprise (WLE) applications, WebLogic Enterprise allows you to define a pool of JDBC database connections. You can use the JDBC connection pooling features in WebLogic Enterprise CORBA Java and WebLogic Enterprise EJB applications.

JDBC connections are expensive resources. Opening and closing them are expensive operations. The JDBC connection pooling feature in WebLogic Enterprise provides efficient use of database connections. Creating a pool of JDBC connections gives WebLogic Enterprise applications ready access to connections that are already open. It removes the overhead of opening a new connection for each database user.

WebLogic Enterprise application developers or system administrators configure the connection pool by using a new section in the application's `UBBCONFIG` file: `JDBC_CONNPOOLS`. WebLogic Enterprise applications use the connection pool at runtime to obtain JDBC connections.

The WebLogic Enterprise software provides connection pooling in its Java infrastructure, to be used on top of different JDBC drivers that integrates with the WebLogic Enterprise administration features.

## About the JDBC Drivers and Connection Pooling

The WebLogic Enterprise software provides the following JDBC drivers:

- WebLogic Enterprise JDBC/XA drivers for Oracle 8.0.5 and 8.1.5 (Type 2)
- `jdbcKona`/Oracle drivers for Oracle 7.3.4, 8.0.5, and 8.1.5 (Type 2)

When you use the `jdbcKona` drivers, you can optionally use the connection pooling feature described in this topic. It is not mandatory that you use connection pooling with the `jdbcKona` drivers. The `jdbcKona` drivers do not support distributed transactions

(also called global, or XA transactions). A local transaction involves updates to a single resource manager, such as a database. A distributed transaction involves updates across multiple resource managers.

WebLogic Enterprise applications that use the JDBC/XA drivers, for local or distributed transactions, must use connection pooling and Oracle 8.0.5 or 8.1.5.

Table 2-1 summarizes the JDBC connection pooling configuration options and requirements.

**Table 2-1 JDBC Connection Pooling Options and Requirements**

<b>JDBC Driver Category</b>	<b>Without JDBC Connection Pooling</b>	<b>With JDBC Connection Pooling</b>
JDBC drivers supporting the JDBC 1.x API. This includes the jdbc/Kona drivers that are included with WebLogic Enterprise: <ul style="list-style-type: none"><li>■ jdbcKona/Oracle 7.3.4</li><li>■ jdbcKona/Oracle 8.0.5 (when ENABLEXA in the application's UBBCONFIG is set to N)</li><li>■ jdbcKona/Oracle 8.1.5 (when ENABLEXA in the application's UBBCONFIG is set to N)</li></ul>	Obtain JDBC connections from <code>DriverManager</code> .	Obtain the WebLogic Enterprise provided <code>DataSource</code> (which wraps around the driver vendor's <code>DriverManager</code> ) from the WebLogic Enterprise JNDI service provider. Then obtain JDBC connections from <code>DataSource</code> .
JDBC drivers supporting JDBC 2.0 Extension API pertaining to Connection Pooling.	Obtain the JDBC driver vendor's <code>DataSource</code> from the WebLogic Enterprise JNDI service provider. Then obtain JDBC connections from <code>DataSource</code> .	Obtain the WebLogic Enterprise provided <code>DataSource</code> (which wraps around the JDBC driver vendor's <code>ConnectionPoolDataSource</code> ) from the WebLogic Enterprise JNDI service provider. Then obtain JDBC connections from <code>DataSource</code> .

**Table 2-1 JDBC Connection Pooling Options and Requirements (Continued)**

JDBC Driver Category	Without JDBC Connection Pooling	With JDBC Connection Pooling
<ul style="list-style-type: none"><li>WebLogic Enterprise JDBC/XA drivers for Oracle 8.0.5 or 8.1.5 (when <code>ENABLEXA</code> in the application's <code>UBBCONFIG</code> is set to <code>Y</code>).</li></ul>	Not applicable. (The WebLogic Enterprise JDBC/XA drivers for Oracle 8.0.5 or 8.1.5 driver must be used in conjunction with WebLogic Enterprise JDBC connection pooling.)	Obtain the WebLogic Enterprise provided <code>DataSource</code> from the WebLogic Enterprise JNDI service provider. Then obtain JDBC connections from <code>DataSource</code> .

# UBBCONFIG Parameters for Connection Pooling

This section describes the application's `UBBCONFIG` file parameters that are related to JDBC connection pooling.

The `JDBCCONNPOOLS` section must be placed after the `SERVERS` section in the configuration file.

The `JDBCCONNPOOLS` section has the following characteristics:

- The entries in the `JDBCCONNPOOLS` section start with the names of connection pools.
- The `SRVID` and `SRVGRP` attributes must refer to a `JavaServer` that is specified in the `SERVERS` section.
- Only the `SRVGRP`, `SRVID`, `MAXCAPACITY`, and `CAPACITYINCR` attributes are required for entries. `TESTTABLE` must be specified if `REFRESH` is specified or if `TESTONRELEASE` or `TESTONRESERVE` are set to `Y`.

**Note:** In the `SERVERS` section, you must also specify `SRVTYPE=JAVA` for the `JavaServer` or `JavaServerXA` to use JDBC connection pooling.

Some attributes are dependent on the version of the JDBC driver.

## Sample UBBCONFIG File for Connection Pooling

Listing 2-1 shows a UBBCONFIG file for a sample multithreaded application that uses the WebLogic Enterprise JDBC/XA driver and connection pooling. Subsequent sections in this topic describe the parameters that are related to JDBC configuration. **Bolded** text is used in the listing to highlight UBBCONFIG section names and parameters that are discussed following the example.

### Listing 2-1 Sample UBBCONFIG for JDBC/XA Bankapp

---

```
*RESOURCES
IPCKEY = 39211
DOMAINID = simple
MASTER = SITE1
MODEL = SHM
LDBAL = N

*MACHINES
trixie
LMID = SITE1
APPDIR = "/myapps/banking"
TUXCONFIG = "/myapps/banking/tuxconfig"
TUXDIR "/wledir"
ULOGPFX "/usr/appdir/logs/ULOG"
MAXACCESSERS = 50

*GROUPS
  SYS_GRP
    LMID      = SITE1
    GRPNO    = 1
  BANK_GROUP1
    LMID      = SITE1
    GRPNO    = 2
  OPENINFO =
"ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=..+DbgFl=0
x7+MaxCur=15+Threads=true"
  TMSNAME   = TMS_ORA
  TMSCOUNT  = 2
```

```
*SERVERS
  DEFAULT:
    RESTART = Y
    MAXGEN  = 5
    .
    .
    .

  JavaServerXA
    SRVGRP = BANK_GROUP1
    SRVID  = 2
    SRVTYPE = JAVA
    CLOPT  = "-A -- -M 10"
    RESTART = N
    .
    .
    .

*MODULES
  BankApp
    SRVGRP = BANK_GROUP1
    SRVID  = 2
    FILE   = "BankApp.jar"
    ARGS   = "TellerFactory_1 bank_pool"

*JDBCConnPOOLS
  bank_pool
    SRVGRP      = BANK_GROUP1
    SRVID       = 2
    DRIVER      = "weblogic.jdbc20.oci815.Driver"
    URL         = "jdbc:weblogic:oracle:Beq-Local"
    PROPS       = "user=scott;password=tiger;server=Beq-Local"
    ENABLEXA    = Y
    INITCAPACITY = 2
    MAXCAPACITY = 10
    CAPACITYINCR = 1
    CREATEONSTARTUP = Y
```

## JDBCConnPOOLS Parameter Values

The following list describes the JDBCConnPOOLS parameters shown in Listing 2-1, “Sample UBBCONFIG for JDBC/XA Bankapp,” on page 2-5. Also described are additional JDBCConnPOOLS parameters that are not shown in the listing.

**Note:** In the `SERVERS` section, you must also specify `SRVTYPE=JAVA` for the `JavaServer` or `JavaServerXA` to use JDBC connection pooling.

- Specify a name for the connection pool. In this example, `bank_pool` is used. This parameter is required and matches the pool name identified in the `ARGS` parameter of the `MODULES` section.
- Use the required `SRVGRP` parameter to identify the server group that will use the connection pool; in this case, `BANK_GROUP1`. `SRVGRP` is a required parameter. The value is a string up to 30 characters. There is no default value.
- Use the `SRVID` parameter to identify the `JavaServer` defined in the `SERVERS` section. `SRVID` is a required parameter. The value is a number from 1 to 30001. There is no default value.
- Use the `DRIVER` parameter to define the class name for the JDBC driver being used. The values are:

- `weblogic.jdbc20.oci815.Driver`
- `weblogic.jdbc20.oci805.Driver`
- `weblogic.jdbc20.oci734.Driver`

The `weblogic.jdbc20.oci815.Driver` value is the driver name for both:

- The WebLogic Enterprise JDBC/XA driver, if you set `ENABLEXA=Y`, for distributed XA transactions
- The `jdbcKona/Oracle` driver for Oracle 8.1.5, if you set `ENABLEXA=N`, for local transactions

The `weblogic.jdbc20.oci805.Driver` value is the **driver name for both**:

- The WebLogic Enterprise JDBC/XA driver, if you set `ENABLEXA=Y`, for distributed XA transactions
- The `jdbcKona/Oracle` driver for Oracle 8.0.5, if you set `ENABLEXA=N`, for local transactions

The WebLogic Enterprise JDBC/XA driver is described in Chapter 1, “Using the WebLogic Enterprise JDBC/XA Drivers.” For this driver, you must use connection pooling and therefore must have a `JDBCPOOLING` section in the application’s `UBBCONFIG` file.

The `jdbcKona/Oracle` driver is described in Chapter 4, “Using the `jdbcKona/Oracle` Drivers.” It is not mandatory that you use JDBC connection pooling with the `jdbcKona/Oracle` driver. However, if your application uses

connection pooling, you must include a `DRIVER` parameter in the application's `UBBCONFIG` file.

- For JDBC drivers that are not JDBC 2.0 compliant, use the `URL` parameter to identify the Universal Resource Locator (URL) that is associated with this driver. If you are using connection pooling with a driver that is not compliant with JDBC 2.0, you must use the `URL` parameter. The URL value is a string up to 256 characters.
- Use the optional `DBNAME` parameter to identify the name of the database. The value is a string up to 30 characters.
- Use the optional `DBUSER` parameter to identify the user account name that will access the database for this WebLogic Enterprise application. The value is a string up to 30 characters.
- Use the optional `DBPASSWORD` parameter to identify the user password for the user account that will access the database for this WebLogic Enterprise application. The value is a string up to 64 characters. This can be specified as clear text or it can be encrypted using the `tmloadcf` command. For details on this option, see “Encrypting DBPASSWORD and PROPS” on page 2-10.
- Use the optional `USERROLE` parameter to identify the SQL role of the user account that will access the database for this WebLogic Enterprise application. The value is a string up to 30 characters.
- Use the optional `DBHOST` parameter to identify the host name of the database server. The value is a string up to 30 characters.
- Use the optional `DBNETPROTOCOL` parameter to identify the network protocol used to communicate with the database. The value is a string up to 30 characters.
- Use the optional `DBPORT` parameter to identify the port number used for database connections. The value is a number up to 65535.
- For JDBC drivers that are not JDBC 2.0 compliant, use the `PROPS` parameter to identify vendor-specific properties for the driver. The value can be a string up to 256 characters. This information can be encrypted. For more information, see “Encrypting DBPASSWORD and PROPS” on page 2-10.
- Use the `ENABLEXA` parameter to indicate whether the connection pool will be used with an XA-compliant driver. The value can be `Y` or `N`. For applications using the WebLogic Enterprise JDBC/XA drivers for Oracle 8.0.5 and 8.1.5, this value must be set to `Y`. The default value is `N`.

- Use the optional `CREATEONSTARTUP` parameter to indicate whether the connection pool is created when the Java server is started. Otherwise, the pool is created when the first request arrives. The value can be `Y` or `N`. The default value is `Y`.
- Use the optional `LOGINDELAY` parameter to indicate the number of seconds to wait between each attempt to open a connection to the database. Some database servers cannot handle multiple requests for connections in rapid succession. This property allows you to build in a small delay to allow the database server to catch up. The value can be any number 0 or greater. The default value is 0.
- Use the optional `INITCAPACITY` parameter to indicate the number of connections initially supported in the connection pool. This should not exceed the value of the related `MAXCAPACITY` parameter. The value for `INITCAPACITY` can be any number 0 or greater. The default value is the value for `CAPACITYINCR`.
- Use the required `MAXCAPACITY` parameter to indicate the maximum number of connections supported in the connection pool. The value is any number 0 or greater. There is no default value for `MAXCAPACITY`.
- Use the required `CAPACITYINCR` parameter to set the number of connections added to the pool when the current limit is exceeded but the maximum capacity is not yet reached. The value is any number 0 or greater. There is no default value.
- Use the `ALLOWSHRINKING` parameter to indicate that the connection pool's number of connections can return to the initial capacity, after expanding to meet demands. The value can be `Y` or `N`. The default value is `N`. The shrinking only closes unused connections.
- Use the `SHRINKPERIOD` parameter to indicate the length of time during which the Java server shrinks the pool to its initial capacity if additional connections are not used. The value is a number in units of minutes.
- Use the `TESTTABLE` parameter to identify the name of the database table that is used to test the validity of connections in the connection pool. The name value can be a string up to 256 characters.

The query `select count(*) from TESTTABLE` is used to test a connection. The table must exist and be accessible to the database user for the connection. This `TESTTABLE` parameter is required if the `REFRESH` parameter is specified, or if the parameter `TESTONRELEASE` or `TESTONRESERVE` is set to `Y`.

- Use the `REFRESH` parameter to specify a time interval for tests performed on the connection pool. This parameter is used in conjunction with the `TESTTABLE` parameter to enable automatic refreshes of connections in pools. The value for the interval is a number in units of minutes. At the specified interval, each unused connection in the pool is tested by executing an SQL query on the connection. If the test fails, the connection's resources are dropped and a new connection is created to replace it.
- Use the optional `TESTONRESERVE` parameter to indicate whether the Java server tests a connection after removing it from the pool and before giving it to the client. The value can be `Y` or `N`. The default value is `N`.
- Use the optional `TESTONRELEASE` parameter to indicate whether the Java server tests a connection before returning it to the connection pool. If all connections in a pool are in use and a client is waiting for a connection, the client will wait longer while the connection is tested. This feature requires that you specify a value for the related `TESTTABLE` parameter (a database table name). The value can be `Y` or `N`. The default value is `N`.
- Use the optional `WAITFORCONN` parameter to indicate whether an application waits indefinitely for a connection if none is currently available. The value can be `Y` or `N`. If the `WAITFORCONN` parameter is `N`, the request for a connection returns to the caller. If the `WAITTIMEOUT` parameter is specified, the default for the `WAITFORCONN` parameter is `N`. If the `WAITTIMEOUT` parameter is not specified, the default for the `WAITFORCONN` parameter is `Y`.
- Use the optional `WAITTIMEOUT` parameter to defines the time interval (in seconds) for an application to wait for a connection to become available. The `WAITFORCONN` and `WAITTIMEOUT` parameters are mutually exclusive. The value for the `WAITTIMEOUT` parameter can be a number 0 or greater, and represents time units in seconds. There is no default value.

## Encrypting DBPASSWORD and PROPS

The `DBPASSWORD` and `PROPS` parameters in the `JDBCConnPools` section specify sensitive data that you may want to encrypt. Values for these attributes can be encrypted in the `UBBCONFIG` file using the `tmloadcf` and `tmunloadcf` utilities.

To store a value for `DBPASSWORD` or `PROPS` in encrypted form, you initially use a text editor to enter a string of five or more continuous asterisks in the parameter value in place of the password in the `UBBCONFIG` file. This string of asterisks is a placeholder for the password. The following is a sample `DBPASSWORD` statement illustrating this:

```
DBPASSWORD="*****"
```

When `tmloadcf` encounters this string of asterisks, it prompts the user to select a password. For example:

```
>tmloadcf -y e:/wle5/samples/atmi/bankapp/xx
DBPASSWORD ("pool2" SRVGRP=GROUP1 SRVID=5):
```

After entering the password, `tmloadcf` stores the password in the `TUXCONFIG` file in encrypted form. If you use `tmunloadcf` to generate a `UBBCONFIG` file, the encrypted password entered is written into the `DBPASSWORD` statement in the `UBBCONFIG` file with `@@` as delimiters. The following is a sample `DBPASSWORD` statement generated by `tmunloadcf`:

```
DBPASSWORD="@@A0986F7733D4@@"
```

When `tmloadcf` encounters an encrypted password in a `UBBCONFIG` file generated by using `tmunloadcf`, it does not prompt the user to create a password. Instead, the `tmloadcf` command uploads the encrypted password back into the system.

**Note:** The `UBBCONFIG` file with the encrypted form of the password may be uploaded back into the system only once; subsequent attempts will fail.

Use of encrypted passwords is only recommended for production environments. Clear-text passwords can be used during application development.

## Displaying Information About JDBC Connection Pools

You can use the `tmadmin printjdbcconnpool` command to report statistics on JDBC connection pools. The data includes the maximum number of connections per pool, the number of connections in use, the number of clients waiting for a connection, and the high-water mark (HWM) or highest number of connections used for a pool.

Listing 2-2 shows the output produced by running the `printjdbcconnpool` command in terse and verbose modes. In terse mode the maximum pool size, the current pool size, and the number of connections currently in use are shown. In verbose mode the number of clients waiting and the high-water mark are also shown.

### Listing 2-2 Sample Output from `tadmin printjdbcconnpool` Command

---

```
>printjdbcconnpool
Pool Name   Grp Name   Srv Id Size Max Size Used
-----
ejbPool     J_SRVGRP   101    1    15    0
Pool        J_SRVGRP   102   10    30    3
```

The following is the verbose mode output for a single connection pool:

```
Pool Name: Pool2
Group ID: J_SRVGRP
Server ID: 102
Driver: (none)
URL: (none)
Database Name: Db
User: leia
Host: SITE1
Password: mypwd
Net Protocol: odbc
Port: 120
Props: (none)
Enable XA: No
Create On Startup: Yes
Pool Size: 10
Maximum Size: 30
Capacity increment: 3
Allow shrinking: Yes
Shrink interval: 10 min(s)
Login delay: 1 sec(s)
Connections in use: 3
Connections awaiting: 0
HWM connections in use: 5
Test table: testtable
Refresh interval: 20 sec(s)
Test conn OnReserve: Yes
Test conn OnRelease: No
```

For example if the high-water mark (HWM) of connections in use is at or close to the maximum size, or connections in use is close to the maximum size and clients are waiting for connections, then you may want to expand the maximum size of the pool. To do this, you must:

- Shut down the WebLogic Enterprise application with the `tmshutdown` command.
- Edit the WebLogic Enterprise application's `UBBCONFIG` file and reconsider the values specified for the particular connection pool's `MAXCAPACITY` parameter in the `JDBCCONNPOOLS` section. You might also want to experiment with the values for the following related `JDBCCONNPOOLS` parameters: `INITCAPACITY`, `CAPACITYINCR`, `ALLOWSHRINKING`, `SHRINKPERIOD`, and `WAITFORCONN` or `WAITTIMEOUT`. See the section "JDBCCONNPOOLS Parameter Values" on page 2-6 for details.
- Use the `tmloadcf` command to create a new binary `TUXCONFIG` version of the application's configuration file.
- Use the `tmboot -y` command to restart the application.

**Note:** Currently, the WebLogic Enterprise software does not support runtime changes to connection pools in running applications.

## T\_JDBCCONNPOOLS MIB Class

The BEA Tuxedo infrastructure supports WebLogic Enterprise features by providing new or enhanced TMIB classes. For JDBC connection pooling, this includes a new `T_JDBCCONNPOOLS` TMIB class. The values that you supply in the a WebLogic Enterprise application's `UBBCONFIG` file are stored in the TMIB classes. The properties defined in these classes are read by the WebLogic Enterprise Java server infrastructure (at boot time) to determine the defined behavior of the application, including the behavior of any connection pools.

System programmers can access the `T_JDBCCONNPOOL` class directly to administer WebLogic Enterprise applications, by using the currently supported TMIB access means. The `T_JDBCCONNPOOL` TMIB class is documented in Section 5 of the *BEA Tuxedo Reference Manual*. This document has been updated and is included in the WebLogic Enterprise online documentation.

# API Characteristics

The WebLogic Enterprise connection pooling feature supports the full JDBC 2.0 Optional Package connection pooling subset, which consists of an application level API and a system level API for interacting with a JNDI Service Provider or other JDBC drivers.

**Note:** The JDBC 2.0 Optional Package was formerly known as the JDBC 2.0 Standard Extension API.

## Application Level API

The JDBC 2.0 application level API provides interfaces for an application to obtain JDBC connections. In the JDBC 2.0 Optional Package, JDBC data sources are implemented by the application server. The data sources serve as JDBC connection factories, through which application users obtain JDBC connections.

The application level API consists of the following interfaces:

- `javax.sql.DataSource`
- `java.sql.Connection`

For JDBC drivers that are compliant with the JDBC 2.0 Optional Package API, the connection is obtained from the driver (which is a reference to the actual `PooledConnection`) and returned to the application directly. However, for a JDBC 1.x driver, the connection object returned to the application is implemented by the WebLogic Enterprise connection pooling module; the connection object is only a reference to the actual database connection returned by the underlying driver.

## System Level API for the JNDI Service Provider

WebLogic Enterprise data sources also implement the following interfaces as an external contract to the WebLogic Enterprise local JNDI Service Provider so that the JNDI Service Provider can interact with it in a standard way:

- `javax.naming.Reference`

- `javax.naming.spi.ObjectFactory`

## System Level API for JDBC drivers

For JDBC 1.x drivers that do not directly support the JDBC 2.0 Optional Package API for connection pooling, the WebLogic Enterprise connection pooling facility provides JDBC 2.0 interface wrappers. Therefore, from the connection pooling module's perspective, it interacts with all drivers with the JDBC 2.0 Optional Package API protocol.

The interfaces supported on behalf of the JDBC 1.x drivers are:

- `javax.sql.ConnectionPoolDataSource`
- `javax.sql.PooledConnection`

The WebLogic Enterprise JDBC connection pooling module also supports the following interface as an external contract to the pooled connections of the JDBC drivers: `javax.sql.ConnectionEventListener`.

## Obtaining Connections from a WebLogic Enterprise Connection Pool

A WebLogic Enterprise application completes the following steps to obtain a JDBC connection from the WebLogic Enterprise connection pool:

1. Obtains a WebLogic Enterprise JNDI implementation

WebLogic Enterprise provides a local JNDI implementation for use within a WebLogic Enterprise JavaServer. Users specify the WebLogic Enterprise initial context factory as the initialization parameter when they get the JNDI initial context, as follows:

```
Context ctx = new InitialContext();
```

For the local WebLogic Enterprise JNDI service provider, you do not have to specify the initial context factory.

2. Obtains the JDBC data source and connection

Data sources are registered in the JNDI namespace by WebLogic Enterprise JavaServers. The name by which it is registered is specified as one of the data source properties in the application's `UBBCONFIG` file. All JDBC data sources are registered in the "jdbc" JNDI naming subcontext of the JNDI root naming context. For example, a data source with the name "EmployeeDB" will be registered with the JNDI name "jdbc/EmployeeDB".

Assume an application needs to obtain a well-known data source called "jdbc/EmployeeDB" from JNDI. The application can get the JDBC connection from the data source, as shown in the following code fragment:

```
/*
 * Assume that it has already obtained JNDI context as in
 * previous step
 */
DataSource ds = (DataSource)ctx.lookup("jdbc/EmployeeDB");
Connection con = ds.getConnection();
```

## An Application's View of the Connection Lifecycle

The `Connection` object returned to the application is only a reference to the underlying database connection. The `Connection` object has the following lifecycle:

- When an application calls `DataSource.getConnection`, the WebLogic Enterprise connection pooling module creates a new `Connection` object on top of an actual database connection that was previously cached or created from the JDBC driver. The `Connection` object is now in the `OPEN` state.
- When the application calls `Connection.close`, or when the connection is implicitly closed by the application server, the connection object is now in the `CLOSED` state, and any subsequent invocation would result in `SQLException`. The underlying database connection is then returned to the connection pool ready to be reused.
- When the connection object is not referenced, it will be subjected to garbage collection.

---

# The DataSource Interface

The `DataSource` implementation in WebLogic Enterprise has the following semantics:

■ `getConnection()`

```
public java.sql.Connection getConnection()  
throws java.sql.SQLException
```

Application users use this `getConnection` method to obtain JDBC connections from the data source. Unlike the JDBC 1.0 `DriverManager.getConnection` API, you do not need to supply the username, password, and URL arguments. The relevant information is made available to the data source via the JDBC data source properties. Applications are responsible for ensuring that the sign-on information is available through appropriate data source properties. That is:

- For JDBC 1.x drivers, through the `driverProps` data source properties
- For JDBC 2.0 drivers, through the `username` and `password` data source properties

You can decide whether to wait for the connection if none is available, and how long to wait for it via the two optional JDBC data source properties:

- `waitForConnection`
- `waitSecondsForConnection`

If the property is not specified, by default `getConnection` will block until a connection is available. If no connection is available after the wait period is specified, an `SQLException` will be thrown, with a message indicating no connection is available.

■ `getConnection(username, password)`

```
public java.sql.Connection getConnection(  
java.lang.String username,  
java.lang.String password)  
throws java.sql.SQLException
```

If the application uses this method to get a connection, the `username` and `password` specified in the arguments will be checked against the values specified in the corresponding JDBC data source properties (which are required for WebLogic Enterprise but not for standard JDBC). If the values match, it behaves the same as the previous method. Otherwise, an `SQLException` will be thrown.

- `getLogWriter()`

```
public java.io.PrintWriter getLogWriter()  
throws java.sql.SQLException
```

Returns the log writer for the data source.

- `setLogWriter()`

```
public void setLogWriter(java.io.PrintWriter out)  
throws java.sql.SQLException
```

The application sets the log writer for the data source using this API.

The WebLogic Enterprise connection pooling facility will intercept the log writer and write the logging information to `ULOG` as well if the Java server `CLOPT` option includes the following parameter: `-jdbclog`.

- `setLoginTimeout`

```
public void setLoginTimeout(int seconds) throws  
java.sql.SQLException
```

Sets the maximum time in seconds that this data source will wait while attempting to connect to a database.

- `getLoginTimeout`

```
public int getLoginTimeout() throws java.sql.SQLException
```

Gets the maximum time in seconds that this data source can wait while attempting to connect to a database.

# 3 Using the jdbcKona Drivers

This chapter covers general guidelines for using the jdbcKona drivers and some vendor-specific notes on each driver. Included at the end of this chapter is a summary of the steps you take, including sample code, to use a JDBC driver in a WebLogic Enterprise Java application.

This topic includes the following sections:

- API Support
- Adding the JAR Files to Your CLASSPATH
- jdbcKona/Oracle Shared Libraries and Dynamic Link Libraries
- Requirements for Making a Connection to a Database Management System (DBMS)
- Support for JDBC Extended SQL
- The JDBC API with WebLogic Extensions
- Implementing a WebLogic Enterprise Java Application Using the jdbcKona Drivers

## API Support

The WebLogic Enterprise 5.1 software supports:

- The JDBC 1.22 API
- The following additional capabilities defined in the JDBC 2.0 Optional Package API:
  - Distributed transactions: the `javax.sql.DataSource` API
  - Connection pooling
  - Java Naming and Directory Interface (JNDI)

New methods that were added in the JDBC 2.0 API, which were not present in JDBC 1.22, are not supported in this release of WebLogic Enterprise. If a WebLogic Enterprise application calls a new JDBC 2.0 method that was not in JDBC 1.22, an `SQLException` will be thrown.

## Platforms Supported by the jdbcKona Drivers

The jdbcKona/Oracle drivers are supported on the following platforms:

- Compaq Tru64 UNIX (Alpha) 4.0F
- HP-UX 11.0
- IBM AIX 4.3.3
- Microsoft Windows 2000 and NT 4.0 SP5
- Sun Solaris 2.6 and 7

For information about any vendor patches required on each platform, and the specific required version of the Java 2 Software Development Kit (SDK), see the WebLogic Enterprise Platform Data Sheets in the WebLogic Enterprise *Installation Guide*.

# Adding the JAR Files to Your CLASSPATH

Be sure to add the WebLogic Enterprise Java ARchive (JAR) files that include the jdbcKona driver classes to your environment.

**Note:** In %TUXDIR% (Windows 2000 or NT) or \$TUXDIR (UNIX), the /udataobj/java/jdbc/jdbckona.jar file used in the WebLogic Enterprise 4.2 and prior WebLogic Enterprise Java releases no longer exists. On systems running the WebLogic Enterprise 5.1 software that will continue to use a jdbcKona driver, update your CLASSPATH to reference the JAR files shown in the next example.

Append the following to your CLASSPATH system environment variable, where TUXDIR is the directory in which you installed the WebLogic Enterprise software:

## Windows 2000 or NT

```
%TUXDIR%\udataobj\java\jdk\M3.jar;%TUXDIR%\udataobj\java\jdk\weblogicaux.jar;
```

## UNIX

```
$TUXDIR/udataobj/java/jdk/M3.jar;$TUXDIR/udataobj/java/jdk/weblogicaux.jar;
```

## jdbcKona/Oracle Shared Libraries and Dynamic Link Libraries

The jdbcKona/Oracle (Type 2) driver calls native libraries that are supplied with the driver. The UNIX libraries (shared object files) are in the \$TUXDIR/lib directory. The Windows DLL files are included in the WebLogic Enterprise Java software kit in the \$TUXDIR\bin directory.

### 3 Using the jdbcKona Drivers

---

Table 3-1 lists the updated names of the jdbcKona/Oracle driver files included with the WebLogic Enterprise Java system.

**Table 3-1 Updated jdbcKona/Oracle Driver Names**

<b>Windows 2000 and NT</b>	<b>Solaris, Compaq Tru64 UNIX, IBM AIX</b>	<b>HP-UX</b>
weblogicoci734.dll	libweblogicoci734.so	libweblogicoci805.sl
weblogicoci805.dll	(Solaris only)	libweblogicoci815.sl
weblogicoci815.dll	libweblogicoci805.so	
	libweblogicoci815.so	

The jdbcKona drivers used in WebLogic Enterprise 4.2 and prior Java releases are removed in the current release. The former driver names were:

- weblogicoci33.dll on Windows NT systems
- weblogicoci33.so on Solaris systems

However, accessing the non-XA jdbcKona drivers with the API used in version 4.2 and prior WebLogic Enterprise releases is still supported; that is, using the `java.sql.DriverManager` API. For example, JDBC applications from WebLogic Enterprise 4.2 should be able to use a WebLogic Enterprise 5.1 jdbcKona/Oracle driver, provided you change the driver class name for the Oracle driver.

The jdbcKona/Oracle driver class names are as follows:

- `weblogic.jdbc20.oci734.Driver`
- `weblogic.jdbc20.oci805.Driver`
- `weblogic.jdbc20.oci815.Driver` (non-XA version)

**Note:** The jdbcKona/Oracle driver for version 8.1.5 is the non-XA version of `weblogic.jdbc20.oci815.Driver`. This non-XA 8.1.5 driver (local transactions only) is used when the driver value is specified and `ENABLEXA=N` is set in the `JDBCCONNPOOLS` section of the application's `UBBCONFIG` file.

For the jdbcKona/Oracle drivers, you also need the vendor-supplied libraries for the database.

# Requirements for Making a Connection to a Database Management System (DBMS)

You need the following components to connect to a DBMS using a jdbcKona driver:

- A database server (Oracle)
- The jdbcKona driver for your database
- The Java 2 software

## Support for JDBC Extended SQL

The Sun Microsystems, Inc. JDBC specification includes *SQL Extensions*, also called *SQL Escape Syntax*. All jdbcKona drivers 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
```

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

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

The following is an example that demonstrates several features of Extended SQL:

```
String insert=
"-- This SQL includes comments and JDBC extended SQL syntax.          \n" +
"insert into date_table values( {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 newline character (\n). The entire Extended SQL sequence, including comments, SQL, and Extended SQL, is placed within double quotes and is passed to the `execute` method of a `Statement` object.

The following Extended SQL is used as part of a `CallableStatement` object:

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

The following 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. The following 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();
```

For a description of Extended SQL, refer to the Sun Microsystems, Inc. Web site, *JDBC Specification 1.2, Chapter 11*.

## The JDBC API with WebLogic Extensions

For the complete set of JDBC API documentation, see the following Web site:

```
http://www.weblogic.com/docs/classdocs/packages.html#jdbc
```

The following packages, classes, interfaces, and WebLogic extensions compose the JDBC API.

**Note:** In the class paths, this section shows `oci815`. However, you would use `oci805` or `oci734`, if you are using Oracle 8.0.5 or Oracle 7.3.4, respectively.

```
Package java.sql
Package java.math

Class java.lang.Object
  Interface java.sql.CallableStatement
    (extends java.sql.PreparedStatement)
  Interface java.sql.Connection
  Interface java.sql.DatabaseMetaData
  Class java.util.Date
    Class java.sql.Date
    Class java.sql.Time
    Class java.sql.Timestamp
  Class java.util.Dictionary
    Class java.util.Hashtable
      (implements java.lang.Cloneable)
    Class java.util.Properties
  Interface java.sql.Driver
  Class java.sql.DriverManager
  Class java.sql.DriverPropertyInfo
  Class java.lang.Math
  Class java.lang.Number
    Class java.math.BigDecimal
    Class java.math.BigInteger
  Interface java.sql.PreparedStatement
    (extends java.sql.Statement)
  Interface java.sql.ResultSet
  Interface java.sql.ResultSetMetaData
  Interface java.sql.Statement
  Class java.lang.Throwable
    Class java.lang.Exception
      Class java.sql.SQLException
      Class java.sql.SQLWarning
      Class java.sql.DataTruncation
  Class java.sql.Types
  Class weblogic.jdbc20.oci815.Connection
    (implements java.sql.Connection)
  Class weblogic.jdbc20.oci815.Statement
    (implements java.sql.Statement)
      Class weblogic.jdbc20.oci815.PreparedStatement
      Class weblogic.jdbc20.oci815.CallableStatement
        (implements java.sql.CallableStatement)
```

The `jdbcKona` drivers provide extensions to JDBC for certain database-specific enhancements. The `jdbcKona` drivers have the following extended classes:

```
Class weblogic.jdbc20.oci815.CallableStatement  
Class weblogic.jdbc20.oci815.Connection  
Class weblogic.jdbc20.oci815.Statement
```

For more information about these extensions, see Chapter 5, “jdbcKona Extensions to the JDBC 1.22 API.”

## Implementing a WebLogic Enterprise Java Application Using the jdbcKona Drivers

This section describes the following steps involved in implementing a simple WebLogic Enterprise Java application that uses a jdbcKona driver to connect to a DBMS:

- Importing Packages
- Setting Properties for Connecting to the DBMS
- Connecting to the DBMS
- Making a Simple SQL Query
- Inserting, Updating, and Deleting Records
- Creating and Using Stored Procedures and Functions
- Disconnecting and Closing Objects

Many of the steps described in this section include code snippets from a comprehensive code example that is provided at the end of this chapter.

For database-specific details on implementing WebLogic Enterprise Java applications using the jdbcKona drivers, see Chapter 4, “Using the jdbcKona/Oracle Drivers.”

## Importing Packages

The classes that you import into your WebLogic Enterprise JavaServer application that uses a jdbcKona driver should include:

```
import java.sql.*;
import java.util.Properties;
```

The jdbcKona drivers implement the `java.sql` interface. You write your WebLogic Enterprise Java application using the `java.sql` classes; the `java.sql.DriverManager` maps the jdbcKona driver to the `java.sql` classes.

You do not import the jdbcKona driver class; instead, you load the driver inside the application. This allows you to select an appropriate driver at run time. You can even decide after the program is compiled what DBMS to connect to.

Included in the WebLogic Enterprise Java software is the latest version of the JDBC API class files. Make sure you do not have any earlier versions of the `java.sql` classes in your CLASSPATH.

You need to import the `java.util.Properties` class only if you use a `Properties` object to set parameters for connecting to the DBMS.

## Setting Properties for Connecting to the DBMS

In the following example, a `java.util.Properties` object sets the parameters for connecting to the DBMS. There are other ways of passing these parameters to the DBMS that do not require a `Properties` object, as in the following snippet:

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

The value for the server property may be vendor-specific; in this example, it is the version 2 alias of an Oracle database running over TCP. You may also add the server name to the URL (see the next section) instead of setting it with the `java.util.Properties` object.

## Connecting to the DBMS

To connect to the DBMS, complete the following steps:

1. Load the proper jdbcKona driver.

The most efficient way to load the jdbcKona driver is to invoke the `Class.forName().newInstance` method with the name of the driver class. This loads and registers the jdbcKona driver, as in the following example for jdbcKona/Oracle for 8.1.5:

```
Class.forName("weblogic.jdbc20.oci815.Driver").newInstance();
```

2. Obtain a JDBC connection.

You request a JDBC connection by invoking the `DriverManager.getConnection` method, which takes as its parameters the URL of the driver and other information about the connection, such as the location of the database and login information.

**Note:** See the section “Obtaining Connections from a WebLogic Enterprise Connection Pool” on page 2-15 for more information about an alternative way of connecting to the DBMS.

Note that both steps describe the jdbcKona driver, but in different formats. The full pathname for the driver is period-separated, while the URL is colon-separated. The following table lists the class paths and URLs for the jdbcKona drivers:

JDBC Driver	Driver Type	Class Pathname	Class URL
jdbcKona/Oracle	Type 2	weblogic.jdbc20.oci734.Driver weblogic.jdbc20.oci815.Driver	jdbc:weblogic:oci

Additional information required to form a database connection varies by DBMS vendor and by whether the jdbcKona driver is of Type 2 or Type 4. There are also a variety of methods for specifying this information in your program.

For full details about the jdbcKona drivers, refer to Chapter 4, “Using the jdbcKona/Oracle Drivers.” For a complete code example, see “Implementing a WebLogic Enterprise Java Application Using the jdbcKona Drivers” on page 3-8.

The connection to the DBMS is handled by the jdbcKona driver. You use both the class name of the driver (in dot-notation) and the URL of the driver (with colons as separators). Class names are case sensitive.

The `Class.forName().newInstance` method loads the driver and registers the driver with the `DriverManager` object.

**Note:** The Sun Microsystems, Inc. *JDBC API Reference* for the `java.sql.Driver` interface recommends simply invoking `Class.forName("driver-class")` to load the driver.

The connection is created with the `DriverManager.getConnection` method, which takes as arguments the URL of the driver and a `Properties` object, as in the following code fragment. The URL is not case sensitive.

```
Class.forName("weblogic.jdbc20.oci815.Driver").newInstance();
Connection conn =
    DriverManager.getConnection("jdbc:weblogic:oracle",
                               props);
conn.setAutoCommit(false);
```

The default transaction mode for JDBC assumes `autocommit` to be true. Setting `autocommit` to false improves performance.

The `Connection` object is an important part of the application. The `Connection` class has constructors for many fundamental database objects that you will use throughout the application. In the examples that follow, you will see the `Connection` object `conn` used repeatedly.

Connecting to the database completes the initial portion of a WebLogic Enterprise Java application, which will be very much the same for any application.

Invoke the `close` method on the `Connection` object as soon as you finish working with the object, usually at the end of a class.

## Making a Simple SQL Query

The most fundamental task in database access is to retrieve data. With a jdbcKona driver, retrieving data is a three-step process:

1. Create a `Statement` object to send an SQL query to the DBMS.

2. Execute the Statement.
3. Retrieve the results into a `ResultSet` object.

In the following code snippet, 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

The following snippet shows three common database tasks: inserting, updating, and deleting records from a database table. We use a JDBC `PreparedStatement` object for these operations; we create the `PreparedStatement` object, then execute the object and close it.

A `PreparedStatement` object (subclassed from `JDBC Statement`) allows you to execute the same SQL over and over again with different values.

`PreparedStatement` objects 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` object to update records. In the following code snippet, 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` object to delete the records that we added and then updated, as in the following snippet:

```
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

You can use a `jdbcKona` driver to create, use, and drop stored procedures and functions. First, we execute a series of `Statement` objects to drop a set of stored procedures and functions from the database, as in the following code snippet:

### 3 Using the jdbcKona Drivers

---

```
Statement stmt = conn.createStatement();
try {stmt.execute("drop procedure proc_squareInt");}
catch (SQLException e) {}
try {stmt.execute("drop procedure func_squareInt");}
catch (SQLException e) {}
try {stmt.execute("drop procedure proc_getresults");}
catch (SQLException e) {}
stmt.close();
```

We use a JDBC `Statement` object to create a stored procedure or function, and then use a JDBC `CallableStatement` object (subclassed from the `Statement` object) with the JDBC "?" syntax to set `IN` and `OUT` parameters. For information about doing this with the jdbcKona/Oracle driver, see Chapter 4, "Using the jdbcKona/Oracle Drivers."

The first two code snippets that follow use the jdbcKona/Oracle driver. Note that Oracle does not natively support binding to "?" values in an SQL statement. Instead, it uses ":1", ":2", and so forth. You can use either syntax in your SQL with the jdbcKona/Oracle driver.

Stored procedure input parameters are mapped to JDBC `IN` parameters, using the `CallableStatement.setxxx` methods, such as `setInt()`, and the "?" syntax of the JDBC `PreparedStatement` object. Stored procedure output parameters are mapped to JDBC `OUT` parameters, using the `CallableStatement.registerOutParameter` methods and the "?" syntax of the JDBC `PreparedStatement` object. A parameter may be both `IN` and `OUT`, which requires both a `setxxx()` and a `registerOutParameter()` invocation to be made on the same parameter number.

In the following code snippet, we use a JDBC `Statement` object to create an Oracle stored procedure; then we execute the stored procedure with a `CallableStatement` object. We use 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();
```

In the following code snippet, we use 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();
```

## Disconnecting and Closing Objects

Close `Statement`, `ResultSet`, `Connection`, and other such objects with their `close` methods after you have finished using them. Closing these objects releases resources on the remote DBMS and within your application. When you use one object to construct another, close the objects in the reverse order in which they were created. For example:

### 3 Using the jdbcKona Drivers

---

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from empno");

    (process the ResultSet)

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

Always close the `java.sql.Connection` as well, usually as one of the last steps in your program. Every connection should be closed, even if a login fails. An Oracle connection will cause a system failure (such as a segment violation) when the finalizer thread attempts to close a connection that you have inadvertently left open. If you do not close connections to log out of the database, you may also exceed the maximum number of database logins. Once a connection is closed, all of the objects created in its context become unusable.

There are occasions in which you will want to invoke the `commit` method to commit changes you have 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 created the `Connection` object for these examples, however, we set `autocommit` to `false`; in this mode, the `Connection` object always has an implicit transaction associated with it, and any invocation to the `rollback` or `commit` methods will end the current transaction and start a new one. Invoking `commit()` before `close()` ensures that all of the transactions are completed before closing the connection.

Just as you close `Statement`, `PreparedStatement`, and `CallableStatement` objects when you have finished working with them, always invoke the `close` method on the connection as final cleanup in your application; enclose the `close` method invocation in a `try {}` block in order to catch exceptions and deal with them appropriately. The final two lines of the example include an invocation to `commit()` and then `close()` to close the connection, as in the following snippet:

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

## Code Example

The following is a sample implementation to give you an overall idea of the structure for a WebLogic Enterprise Java application that uses a `jdbcKona` driver to access a DBMS. The code example shown here includes retrieving data, displaying metadata,

inserting, deleting, and updating data, and stored procedures and functions. Note the explicit invocations to `close()` for each JDBC-related object, and note also that we close the connection itself in a `finally {}` block, with the invocation to `close()` wrapped in a `try {}` block.

```
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");

            Class.forName("weblogic.jdbc20.oci815.Driver").newInstance();
            Connection conn =
                DriverManager.getConnection("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++) {
```

### 3 Using the *jdbcKona Drivers*

---

```
        System.out.println("Column Name: "      + md.getColumnNames(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 {stmtDrop.execute("drop procedure func_squareInt"); }
    catch (SQLException e) {}
    try {stmtDrop.execute("drop procedure proc_getresults"); }
    catch (SQLException e) {}
    stmtDrop.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();
```

### 3 Using the jdbcKona Drivers

---

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

---

# 4 Using the jdbcKona/Oracle Drivers

This chapter provides guidelines for using the jdbcKona/Oracle Type 2 drivers. For general notes about and an example of using the jdbcKona drivers, see Chapter 3, “Using the jdbcKona Drivers.”

This topic includes the following sections:

- Data Type Mapping
- Connecting a jdbcKona/Oracle Driver to an Oracle DBMS
- Waiting for Oracle DBMS Resources
- Autocommit
- Using Oracle Blobs
- Support for Oracle Array Fetches
- Using Stored Procedures
- DatabaseMetaData Methods
- jdbcKona/Oracle and the Oracle NUMBER Column

# Data Type Mapping

Mapping of types between Oracle and the jdbcKona/Oracle drivers are provided in the following table.

Oracle	jdbcKona/Oracle Drivers
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

**Note:** When the `PreparedStatement.setBoolean` method is invoked, this method converts a `VARCHAR` type to "1" or "0" (string), and it converts a `NUMBER` type to 1 or 0 (number).

# Connecting a jdbcKona/Oracle Driver to an Oracle DBMS

To make a DBMS connection, complete the following steps:

**Note:** See the section “Obtaining Connections from a WebLogic Enterprise Connection Pool” on page 2-15 for more information about an alternative way of connecting to the DBMS.

1. Load the proper jdbcKona driver.

The most efficient way to do this is to invoke the `Class.forName().newInstance()` method with the name of the driver class, which properly loads and registers the jdbcKona driver. For example:

```
Class.forName("weblogic.jdbc20.oci734.Driver").newInstance();
```

2. Request a JDBC connection by invoking the `DriverManager.getConnection` method, which takes as its parameters the URL of the driver and other information about the connection.

Both steps describe the jdbcKona driver, but in a different format. The full package name is period-separated, and the URL is colon-separated. The URL must include at least `jdbc:weblogic:oracle`, and may include other information, including server name and database name.

There are several variations on this basic pattern, which are described here for Oracle. For a full code example, see Chapter 3, “Using the jdbcKona Drivers.”

## Method 1

The simplest way to connect to an Oracle DBMS is by passing the URL of the driver that includes the name of the server, along with a username and a password, as arguments to the `DriverManager.getConnection` method, as in the following jdbcKona/Oracle example:

```
Class.forName("weblogic.jdbc20.oci734.Driver").newInstance();  
Connection conn =
```

```
DriverManager.getConnection("jdbc:weblogic:oracle:DEMO",
                            "scott",
                            "tiger");
```

In the example, DEMO is the version 2 alias of an Oracle database. Note that invoking the `Class.forName().newInstance()` method properly loads and registers the driver.

## Method 2

You can also pass a `java.util.Properties` object with parameters for connection as an argument to the `DriverManager.getConnection` method. The following example shows how to connect to the DEMO database:

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

Class.forName("weblogic.jdbc20.oci734.Driver").newInstance();
Connection conn =
    DriverManager.getConnection("jdbc:weblogic:oracle",
                               props);
```

If you do not supply a server name (DEMO in the preceding example), the system looks for an environment variable (ORACLE\_SID in the case of Oracle). You can also add the server name to the URL, using the following format:

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

When you use the preceding format, you do not need to provide a "server" property.

## Other Properties You Can Set for the jdbcKona/Oracle Driver

There are other properties that you can set for the jdbcKona/Oracle driver, which are covered later in this document. The jdbcKona/Oracle driver also allows setting a property `-- allowMixedCaseMetaData --` to the boolean `true`. This property sets up the connection to use mixed case letters in invocation to `DatabaseMetaData` methods. Otherwise, Oracle defaults to uppercase letters for database metadata.

The following is an example of setting 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");

Connection conn =
    DriverManager.getConnection("jdbc:weblogic:oracle",
                                props);
```

If you do not set this property, the jdbcKona/Oracle driver defaults to the Oracle default, which uses uppercase letters for database metadata.

## General Notes

Always invoke the `Connection.close` method to close the connection when you have finished working with it. Closing objects releases resources on the remote DBMS and within your application, as well as being good programming practice. Other jdbcKona objects on which you should invoke the `close` method after final use include:

- `Statement` (`PreparedStatement`, `CallableStatement`)
- `ResultSet`

## Waiting for Oracle DBMS Resources

The jdbcKona/Oracle driver supports the Oracle `oopt()` C API, which allows a client to wait until resources become available. The Oracle C function sets options in cases where requested resources are not available; for example, whether to wait for locks.

You can set whether a client waits for DBMS resources, or receives an immediate exception.

**Note:** In the driver class path examples, the format is:

```
weblogic.jdbc20.ociXXX.Driver
```

Where xxx is the version of the Oracle database: 734 for version 7.3.4, or 805 for version 8.0.5, or 815 for version 8.1.5.

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

Class.forName("weblogic.jdbc20.oci734.Driver").newInstance();

// You must cast the Connection as a
// weblogic.jdbc20.ociXXX.Connection
// to take advantage of this extension
Connection conn =
    (weblogic.jdbc.oci734.Connection)
        DriverManager.getConnection("jdbc:weblogic:oracle", props);

// After constructing the Connection object, immediately call
// the waitOnResources method
conn.waitOnResources(true);
```

The `waitOnResources()` method can cause several error return codes while waiting for internal resources that are locked for short durations.

To take advantage of this feature, you must first cast your `Connection` object as a `weblogic.jdbc20.oci[version].Connection` object, and then invoke the `waitOnResources` method (where `[version]` is 734, or 805, or 815).

This functionality is described in section 4-97 of *The OCI Functions for C*, published by Oracle Corporation.

## Autocommit

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

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

---

# Using Oracle Blobs

The `jdbcKona/Oracle` driver supports two new properties to support Oracle Blob chunking:

- `weblogic.oci.insertBlobChunkSize`

This property affects the buffer size of input streams bound to a `PreparedStatement` object. Blob chunking requires an Oracle 7.3.x or higher Oracle Server; to use this property, you must be connected to an Oracle Server that supports this feature.

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 0 (zero), which means that Blob chunking is turned off.

- `weblogic.oci.selectBlobChunkSize`

This property sets the size of output streams associated with a `JDBC ResultSet` object. The mechanism for piecewise selects does not have the same use restrictions as that for Blob inserts, so this property is set to 65534 by default. It is not necessary to turn this property off.

Set this property to the size of the desired output stream, in bytes.

## Support for Oracle Array Fetches

With WLE Java, the `jdbcKona/Oracle` driver supports Oracle array fetches. With this feature support, invoking the `ResultSet.next` method the first time gets an array of rows and stores it in memory, rather than retrieving a single row. Each subsequent invocation of the `next` method reads a row from the rows in memory until they are exhausted, and only then does the `next` method 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. The following is an example of setting this property to 300, which means that invocations to the `next` method hit the database only 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");

Class.forName("weblogic.jdbc20.oci734.Driver").newInstance();
Connection conn =
    DriverManager.getConnection("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. Tune your application for the best balance between performance and client resources, depending upon your network configuration and your application.

If any columns in a `SELECT` statement are of type `LONG`, the cache size will be temporarily reset to 1 (one) for the `ResultSet` object associated with that select statement.

# Using Stored Procedures

The following sections describe how to use stored procedures:

- Syntax for Stored Procedures in the jdbcKona/Oracle Driver
- Binding a Parameter to an Oracle Cursor
- Using `CallableStatement`

---

## Syntax for Stored Procedures in the jdbcKona/Oracle Driver

The syntax for stored procedures in Oracle was altered in the jdbcKona/Oracle driver examples to match the JDBC specification. (All of the examples also show native Oracle SQL, commented out, just above the correct usage; the native Oracle syntax works as it did in the past.) You can read more about stored procedures for the jdbcKona drivers in Chapter 3, “Using the jdbcKona Drivers.”

Note that Oracle does not natively support binding to "?" values in an SQL statement. Instead it uses ":1", ":2", and so forth. We allow you to use either in your SQL with the jdbcKona/Oracle driver.

## Binding a Parameter to an Oracle Cursor

BEA Systems, Inc. has created an extension to JDBC, `weblogic.jdbc20.oci[version].CallableStatement`, that allows you to bind a parameter for a stored procedure to an Oracle cursor (where [version] is 734, or 805, or 815. You can create a `JDBC ResultSet` object with the results of the stored procedure. This allows you to return multiple `ResultSet` objects in an organized way. The `ResultSet` objects are determined at run time in the stored procedure. An example procedure follows.

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, construct `CallableStatement` objects with the stored procedures and register the output parameter as data type `java.sql.Types.OTHER`. When you retrieve the data into a `ResultSet` object, use the output parameter index as an argument for the `getResultSet` method. For example:

```
weblogic.jdbc20.oci734.CallableStatement cstmt =
    (weblogic.jdbc20.oci734.CallableStatement)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();

weblogic.jdbc20.oci734.CallableStatement cstmt2 =
    (weblogic.jdbc20.oci734.CallableStatement)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();

weblogic.jdbc20.oci734.CallableStatement cstmt3 =
    (weblogic.jdbc20.oci734.CallableStatement)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);

```

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

## Using CallableStatement

The default length of a string bound to an `OUTPUT` parameter of a `CallableStatement` object is 128 characters. If the value you assign to the bound parameter exceeds that length, you 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. The following is a code example that binds a `VARCHAR` that will never be larger than 256 characters:

```

CallableStatement cstmt =
    conn.prepareCall("BEGIN testproc(?); END;");

cstmt.registerOutputParameter(1, Types.VARCHAR, 256);
cstmt.execute();
System.out.println(cstmt.getString());
cstmt.close();

```

## DatabaseMetaData Methods

DatabaseMetaData is implemented in its entirety in the jdbcKona/Oracle driver. There are some variations that are specific to Oracle, which are as follows:

- 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.

## jdbcKona/Oracle and the Oracle 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.

The jdbcKona/Oracle driver reliably converts the values in a column to the Java type requested when a WLE Java 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. The `jdbcKona/Oracle` driver guarantees to return a Java object that 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 the `jdbcKona/Oracle` driver must decide what sort of object to return based on each value. This means that one `ResultSet` object may return multiple Java types from the `getObject` method for a given `NUMBER` column. A table full of integer values may all be returned as `Integer` from the `getObject` method, 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 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 the `jdbcKona/Oracle` driver always returns a Java object appropriate for the given precision and scale, regardless of the values shown in the following table. The following table shows the Java objects returned for each qualified `NUMBER` column.

<b>Column Definition</b>	<b>Returned by <code>getObject()</code></b>
<code>NUMBER(P &lt;= 9)</code>	<code>Integer</code>
<code>NUMBER(P &lt;= 18)</code>	<code>Long</code>
<code>NUMBER(P &gt;= 19)</code>	<code>BigDecimal</code>
<code>NUMBER(P &lt;=16, S &gt; 0)</code>	<code>Double</code>
<code>NUMBER(P &gt;= 17, S &gt; 0)</code>	<code>BigDecimal</code>



# 5 jdbcKona Extensions to the JDBC 1.22 API

This chapter describes the following jdbcKona extensions to the JDBC API:

- Class `weblogic.jdbc20.oci[version].CallableStatement`
- Class `weblogic.jdbc20.oci[version].Connection`
- Class `weblogic.jdbc20.oci[version].Statement`

**Note:** In the previous list, `oci[version]` refers to the Oracle version number (734, 805, or 815). The samples in this chapter show `oci734`. For example:

```
Class weblogic.jdbc20.oci734.CallableStatement
```

However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

For complete details on this JDBC API, refer to the following Web site:

```
http://www.weblogic.com/docs51/classdocs/jdbcdrivers.html
```

If this URL changes and you cannot locate this BEA WebLogic JDBC API, please go to <http://e-docs.bea.com/>. On that page, click the link for the BEA WebLogic Server™ (WLS) documentation. On the WLS page, click the JDBC link in the Quick Links box.

# Class CallableStatement

Class `weblogic.jdbc20.oci734.CallableStatement` contains `jdbcKona` extensions to JDBC to support the use of cursors as parameters in `CallableStatement` objects.

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

The `CallableStatement` class:

- Extends the `PreparedStatement` class
- Implements the `CallableStatement` interface
- Has the following inheritance hierarchy:

```
java.lang.Object
|
+----weblogic.jdbc20.oci734.Statement
|
+----weblogic.jdbc20.oci734.PreparedStatement
|
+----weblogic.jdbc20.oci734.CallableStatement
```

- Has the `getResultSet` method

---

**weblogic.jdbc20.oci734.CallableStatement.getResultSet**

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

**Synopsis** Returns a `ResultSet` object from a stored procedure where the specified parameter has been bound to an Oracle cursor. Register the output parameter with the `registerOutputParameter` method, using `java.sql.Types.OTHER` as the data type.

**Java Mapping** `public ResultSet getResultSet(int parameterIndex) throws SQLException`

**Parameters** `parameterIndex`  
This parameter is an index into the set of parameters for the stored procedure.

**Throws** `SQLException`  
This exception is thrown if the operation cannot be completed.

## Class Connection

This section describes only the `jdbcKona` extension to JDBC that accesses the Oracle OCI C Function `oopt()`. Other information about this class is in the description for class `java.sql.Connection`. A `Connection` object is usually constructed as a `java.sql.Connection` class. To use this extension to JDBC, you must explicitly cast your `Connection` object as a `weblogic.jdbc20.oci734.Connection` class.

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

The public `Connection` class:

- Extends the `Object` class
- Implements the `Connection` interface
- Has the following inheritance hierarchy:

```
java.lang.Object
|
+----weblogic.jdbc20.oci734.Connection
```

- Has the `waitOnResources` method

---

## weblogic.jdbc20.oci734.Connection.waitOnResources

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

**Synopsis** Use this method to access the Oracle `oopt()` function for C (see section 4-97 of *The OCI Functions for C*). The Oracle C function sets options in cases where requested resources are not available; for example, whether to wait for locks.

When the argument to this method is true, this `jdbcKona` extension to JDBC sets this option so that your program will receive an error return code whenever a resource is requested but is unavailable. Use of this method can cause several error return codes while waiting for internal resources that are locked for short durations.

**Java Mapping** `public void waitOnResources(boolean val)`

**Parameters** `val`

This parameter is set to true if the connection should wait on resources.

## Class `weblogic.jdbc20.oci734.Statement`

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

This class contains `jdbcKona` extensions to JDBC to support parsing of SQL statements and adjusting of the fetch size. Only those methods are documented here.

The `weblogic.jdbc20.oci734.Statement` class:

- Extends the Object base class
- Has the following inheritance hierarchy:

```
java.lang.Object
|
+----weblogic.jdbc20.oci734.Statement
```

- Has the following methods:
  - `fetchsize`
  - `parse`

## **weblogic.jdbc20.oci734.Statement.fetchsize**

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

**Synopsis** Allows tuning of the size of prefetch array used for Oracle row results. Oracle provides the means to do data prefetch in batches, which decreases network traffic and latency for row requests.

The default batch size is 100. Memory for 100 rows is allocated in the native stack for every query. For queries that need fewer rows, this size can be adjusted appropriately. This saves on the swappable image size of the application and will benefit performance if only as many rows as needed are fetched.

**Java Mapping** `public void fetchSize(int size)`

**Parameters** `size`

This parameter specifies the number of rows to be prefetched.

### **weblogic.jdbc20.oci734.Statement.parse**

**Note:** In the class paths, this section shows `oci734`. However, you would use `oci805` or `oci815`, if you are using Oracle 8.0.5 or Oracle 8.1.5, respectively.

**Synopsis** Allows tuning of the size of prefetch array used for Oracle row results. Oracle provides the means to do data prefetch in batches, which decreases network traffic and latency for row requests.

The default batch size is 100. Memory for 100 rows is allocated in the native stack for every query. For queries that need fewer rows, this size can be adjusted appropriately. This saves on the swappable image size of the application and will benefit performance if only as many rows as needed are fetched.

**Java Mapping** `public int parse(String sql) throws SQLException`

**Parameters** `sql`  
This parameter is the SQL statement to be verified.

**Throws** `SQLException`  
This exception is thrown if the operation cannot be completed.

---

# Index

## A

- ALLOWSHRINKING parameter 2-9
- array fetches
  - support for 4-7
- autocommit
  - using with Oracle 4-6

## B

- Blobs
  - Oracle 4-7

## C

- CallableStatement class 4-11
  - API for WebLogic extension to 5-2
- CAPACITYINCR parameter 2-9
- class pathname
  - for DBMS connection 3-10
- CLASSPATH 1-3, 3-3
- closing objects 3-15
- connecting to a DBMS 3-10
  - requirements for making 1-5, 3-5
- Connection class
  - API for WebLogic extension to 5-4
- connection pooling 2-1, 2-2
- connection pools
  - displaying data about 2-11
- CREATEONSTARTUP parameter 2-9
- customer support contact information ix

## D

- data type mapping 4-2
- database management system
  - see DBMS
- DatabaseMetaData methods
  - using 4-4
  - variations specific to Oracle 4-12
- DataSource interface 2-17
  - getConnection 2-17
  - getLoginTimeout 2-18
  - getLogWriter 2-18
  - setLoginTimeout 2-18
  - setLogWriter 2-18
  - using with JDBC/XA driver 1-6
- DBHOST parameter 2-8
- DBMS connections
  - class pathname 3-10
  - making 3-10
  - requirements for making 1-5, 3-5
  - setting properties for 3-9
- DBNAME 2-8
- DBNETPROTOCOL parameter 2-8
- DBPASSWORD parameter 2-8
- DBPORT parameter 2-8
- DBUSER parameter 2-8
- distributed transactions
  - JDBC/XA driver 1-1
- DLLs
  - for jdbcKona/Oracle 3-3
- documentation, where to find it viii
- DRIVER parameter 2-7

---

## E

- ENABLEXA parameter 2-8
- encrypting passwords 2-10
- Extended SQL
  - JDBC support for 3-5

## F

- fetchsize method 5-7

## G

- getConnection method 2-16, 4-3
- getResultSet method 5-3

## I

- implementing, using jdbcKona drivers 3-8
- importing packages 3-9
- INITCAPACITY parameter 2-9
- InitialContext method 2-15

## J

- Java 2 1-3, 3-2
- java.math 3-6
- java.sql 3-6
- javax.sql.ConnectionEventListener 2-15
- javax.sql.ConnectionPoolDataSource 2-15
- JDBC
  - API 3-6
  - Extended SQL
    - support for 3-5
  - jdbcKona extensions to 3-6
  - supported version 1-1, 3-1
- JDBC connection pooling 2-1, 2-2
  - API 2-14
  - application level API 2-14
  - connection lifecycle 2-16
  - system level API for JDBC drivers 2-15
  - system level API for JNDI 2-14

- JDBC connection pools
  - attributes of 2-6
  - encrypting passwords used with 2-10
- JDBC/XA driver
  - using 1-1
- JDBCConnPools
  - ALLOWSHRINKING parameter 2-9
  - CAPACITYINCR parameter 2-9
  - CREATEONSTARTUP parameter 2-9
  - DBHOST parameter 2-8
  - DBNAME parameter 2-8
  - DBNETPROTOCOL parameter 2-8
  - DBPASSWORD parameter 2-8
  - DBPORT parameter 2-8
  - DBUSER parameter 2-8
  - DRIVER parameter 2-7
  - ENABLEXA parameter 2-8
  - INITCAPACITY parameter 2-9
  - LOGINDELAY parameter 2-9
  - MAXCAPACITY parameter 2-9
    - parameters 2-6
    - pool name 2-7
  - PROPS parameter 2-8
  - REFRESH parameter 2-10
  - sample section 2-5
  - SHRINKPERIOD parameter 2-9
  - SRVGRP parameter 2-7
  - SRVID parameter 2-7
  - TESTONRELEASE parameter 2-10
  - TESTONRESERVE parameter 2-10
  - TESTTABLE parameter 2-9
  - URL parameter 2-8
  - USERROLE parameter 2-8
  - WAITFORCONN parameter 2-10
  - WAITTIMEOUT parameter 2-10
- jdbcKona drivers
  - implementing in a WLE Java application 3-8
  - JAR file 1-3, 3-3
  - making an SQL query with 3-11
  - platforms supported on 1-3, 3-2

---

- sample code using 3-16
- support for JDBC Extended SQL 3-5
- jdbcKona/Oracle drivers
  - and array fetches 4-7
  - and Blob chunking 4-7
  - and Oracle NUMBER column 4-12
  - closing connections with 4-5
  - connecting to Oracle DBMS 4-3
  - DLLs 3-3
  - shared libraries 3-3
  - using stored procedures in 4-8
- JDK 1.2
  - See Java 2

## L

- LOGINDELAY parameter 2-9

## M

- MAXCAPACITY parameter 2-9

## N

- newInstance method 4-3
- NUMBER column 4-12

## O

- objects
  - disconnecting and closing 3-15
- Oracle cursor 4-9
- Oracle oopt() C function
  - accessing 5-5
  - API 4-5
- Oracle rows 5-7

## P

- packages
  - importing 3-9
- parameter

- binding to an Oracle cursor 4-9
- parse method 5-8
- passwords
  - encrypting 2-10
- pooling
  - database connections 2-1
  - JDBC connections 2-2
- PreparedStatement class 3-12
- printing product documentation viii
- printjdbcconnpool option 2-11
- properties
  - setting for a DBMS connection 3-9
- Properties object 4-4
- PROPS parameter 2-8

## R

- records
  - inserting, updating, and deleting 3-12
- REFRESH parameter 2-10
- related information viii
- resources
  - waiting for Oracle DBMS 4-5
- ResultSet class 4-9
- ResultSet object
  - returning from stored procedure 5-3

## S

- shared libraries
  - for jdbcKona/Oracle 3-3
- shrinking connection pools 2-9
- SHRINKPERIOD parameter 2-9
- Solaris 1-3, 3-2
- SQL query
  - making with a jdbcKona driver 3-11
- Statement class
  - API for WebLogic extension to 5-6
- stored procedures
  - creating and using 3-13
  - returning ResultSet object from 5-3

---

using in jdbcKona/Oracle 4-8  
support  
technical ix

## **T**

T\_JDBCConnPOOLS TMIB class 2-13  
TESTONRELEASE parameter 2-10  
TESTONRESERVE parameter 2-10  
TESTTABLE parameter 2-9  
tmadmin command  
    printjdbcconnpool option 2-11

## **U**

UBBCONFIG  
    JDBCConnPOOLS sample 2-5  
    parameters for connection pooling 2-4  
    sample file for connection pooling 2-5  
USERROLE parameter 2-8

## **W**

WAITFORCONN parameter 2-10  
waitOnResources method 5-5  
WAITTIMEOUT parameter 2-10  
WebLogic extensions  
    Connection class 5-4  
    to CallableStatement class 5-2  
    to JDBC (list) 3-6  
    to Statement class 5-6  
weblogic.jdbc20.oci734.Driver 2-7  
weblogic.jdbc20.oci804.Driver 2-7  
weblogic.jdbc20.oci815.Driver 2-7, 3-4  
Windows NT 4.0 1-3, 3-2  
WLE Java application 3-8

## **X**

XA  
    JDBC/XA driver 1-1