BEA WebLogic Enterprise

Using the JDBC Drivers

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Using the JDBC Drivers

Document Edition	Date	Software Version
5.0	December 1999	BEA WebLogic Enterprise 5.0

About This Document

This document contains programming and reference information for the JDBC drivers that are provided with the BEA WebLogic Enterprise (sometimes referred to as WLE) software.

This document covers the following topics:

- Chapter 1, “Using the WLE JDBC/XA Driver.”
- Chapter 2, “Using JDBC Connection Pooling.”
- Chapter 3, “Using the jdbcKona Drivers.”
- Chapter 4, “Using the jdbcKona/Oracle Driver.”
- Chapter 5, “Using the jdbcKona/ MSSQLServer4 Driver.”
- Chapter 6, “jdbcKona Extensions to the JDBC 1.22 API.”

What You Need to Know

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

e-docs Web Site

The BEA WLE product documentation is available on the BEA 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.beasys.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 WLE 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 WLE documentation Home page, click the PDF Files button and select the document you want to print.

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

Related Information

Before installing the BEA WLE 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 WLE Bibliography at http://edocs.beasys.com/.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WLE 5.0 release.

If you have any questions about this version of BEA WLE, or if you have problems installing and running BEA WLE, contact BEA Customer Support through BEA WebSupport at **www.beasys.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
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.

Convention	Item
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: 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.
	Example:
	`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:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contents

2. Using JDBC Connection Pooling

3. Using the jdbcKona Drivers

4. Using the jdbcKona/Oracle Driver

5. Using the jdbcKona/ MSSQLServer4 Driver

1 Using the WLE JDBC/XA Driver

You can use the WebLogic Enterprise (WLE) JDBC/XA driver to make local or distributed connections to Oracle 8i databases. You can use this driver with WLE CORBA Java and WLE J2EE (EJB and RMI) applications.

This topic includes the following sections:

- Before You Begin
  - Platforms Supported by the JDBC/XA Driver
  - 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 WLE JDBC/XA driver, see Transactions and the WLE JDBC/XA Driver. 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

WLE applications using the WLE JDBC/XA for Oracle 8i driver can perform local transactions as well as distributed (also called global) transactions. A local transaction involves updates to a single Resource Manager, such as a database. A distributed transaction involves updates across multiple Resource Managers.

Read the following topics before you start using the WLE JDBC/XA for Oracle 8i driver:

- Platforms Supported by the JDBC/XA Driver
- 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

Platforms Supported by the JDBC/XA Driver

The following table lists the platforms supported by the WLE JDBC/XA driver.

Operating System and Version	Java 2 Software Development Kit (SDK)	DBMS	Client Libraries
Windows NT 4.0 (SP4)	Java 2 SDK 1.2.2	Oracle 8i or higher	Oracle 8i
Solaris 2.6 and 7.0	Java 2 SDK 1.2.1, Production release	Oracle 8i or higher	Oracle 8i
HP-UX 11.0	Java 2 SDK 1.2.1	Oracle 8i or higher	Oracle 8i

Adding the JAR Files to Your CLASSPATH

Be sure to add the WLE JAR files that include the JDBC/XA driver classes to your environment. You can do this by appending the following to your `CLASSPATH` system environment variable, where `TUXDIR` is the directory in which you installed the WLE software:

On UNIX Systems:

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

On NT Systems:

```
%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 setup 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 WLE software:

On UNIX Systems:

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

On NT Systems:

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

Shared Libraries and Dynamic Link Libraries

The JDBC/XA 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 WLE software kit in the `%TUXDIR%\bin` directory.

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

Windows NT/98/95	Solaris	HP-UX
`weblogicoci815.dll`	`libweblogicoci815.so`	`libweblogicoci815.sl`

For the WLE JDBC/XA driver, the driver class name is `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 WLE environment, this step is done by setting parameters in the WLE application's UBBCONFIG file. For more information, see the section "Setting Data Source Properties" on page 1-5.

For the JDBC/XA driver, you also need the Oracle-supplied version 8i 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 8i database server
- The WLE JDBC/XA for Oracle 8i driver
- The Java 2 Software Developer Kit (Java 2 SDK)
  - For NT systems, use the Java 2 SDK 1.2.2
  - For UNIX systems, use the Java 2 SDK 1.2.1

About the Sample Code

In addition to the supported sample applications that are provided with the WLE 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 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 WLE 5.0 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 WLE. If a WLE 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 WLE environment, you set the data source properties separate from the WLE 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 runtime 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 WLE application's Java server is booted, its infrastructure will read the properties from TMIB and initialize the connection pools.

The set up 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 WLE 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 8i
- Define the Database Open Information
- Define JavaServerXA Parameters, Including the Connection Pool Name
- 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 rmname [-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 (RM) associated with this server. If the JavaServerXA is being deployed in multithreaded mode, you must ensure that the RM contains values for Oracle 8i . Attempting to deploy a multithreaded JavaServerXA that is being linked with an RM other than Oracle 8i 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 NT systems. On UNIX systems, each entry in this file is of the form `rmname:rmstructure_name:library_names`. On 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 8i.

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 WLE libraries and include files to use when compiling the server application.

`LD_LIBRARY_PATH` (UNIX Systems)
: Indicates which directories contain shared objects to be used by the compiler, in addition to the WLE shared objects. A colon (:) is used to separate the list of directories.

`LIB` (Windows NT systems)
: Indicates a list of directories within which to find libraries. A semicolon (;) is used to separate the list of directories.

Portability

The `buildXAJS` command is supported in UNIX and NT systems. It is not supported on client-only WLE 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). In the current release, Oracle 8i is the RM that you can use with the JDBC/XA driver. The files that result from the `buildtms` command need to be installed in `$TUXDIR/bin` directory.

Syntax

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

Examples

The following examples build transaction manager server load modules for Oracle 8i.

On UNIX Systems

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

On NT Systems

```
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 file name for the output load module.

`-r rmname`
Specifies the resource manager (RM) associated with this server. If the JavaServerXA is being deployed in multithreaded mode, you must ensure that the RM contains values for Oracle 8i . Attempting to deploy a multithreaded JavaServerXA that is being linked with an RM other than Oracle 8i 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 NT systems. On UNIX systems, each entry in this file is of the form `rmname:rmstructure_name:library_names`. On 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 8i.

Portability

The `buildtms` command is supported in UNIX and NT systems. It is not supported on client-only WLE 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 8i 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 8i in the previous `buildtms` step.

Define JavaServerXA Parameters, Including the Connection Pool Name

In the `SERVERS` section of the application's UBBCONFIG file, define parameters to indicate how this WLE application will use JavaServerXA. The parameters include the name of the connection pool. For example:

Each JavaServerXA can only host WLE JDBC connection pools that connect to one Resource Manager. The current release supports the Oracle 8i XA Resource Manager only.

These parameters are 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.

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. Also notice the `bank_pool` value to give a handle for a JDBC connection pool. Also, you must specify `SRVTYPE=JAVA` for the JavaServer or JavaServerXA to use JDBC connection pooling.

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 BankApp.jar TellerFactory_1 bank_pool"
RESTART = N
```

The `-M 10` parameter 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 `JDBCCONNPOOLS` 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 `JDBCCONNPOOLS` section was added to the WLE product in version 5.0. Listing 1-3 shows a sample section.

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

```
*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
```

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 used as a command line (`CLOPT`) parameter in the `SERVERS` section for JavaServerXA. The use of `bank_pool` forms the association between the JavaServerXA that will run in `BANK_GROUP1` and this connection pool.

For the WLE JDBC/XA driver, set the `DRIVER` parameter value to the value shown: `weblogic.jdbc20.oci815.Driver`. Also, the `ENABLEXA` parameter must be set to `Y`.

See Chapter 2, "Using JDBC Connection Pooling" for information about the other `JDBCCONNPOOLS` parameters. They include settings for the initial and runtime 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 WLE 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 factory finder
- Establishes connections to the database

The two arguments are:

- `Teller_Factory_1`
- `JdbcConnPoolName`

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("TellFactoryName JdbcConnPoolName");
 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("tellerFName: " + tellerFName );
TP.userlog("JDBC connection pool name: " + pool_name);
```

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

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 WLE JDBC applications that include distributed (XA) transactions and connection pooling must use a `DataSource` object to get connections.

For example, the following code fragement 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 WLE JDBC/XA Driver,” descibes the JDBC/XA driver provided with the WLE software and introduces the use of connection pooling with that driver. This chapter explains how to use connection pooling with any JDBC driver supported by the WLE software. This includes the WLE JDBC/XA driver and the jdbcKona drivers that are documented in Chapter 4, “Using the jdbcKona/Oracle Driver” and Chapter 5, “Using the jdbcKona/ MSSQLServer4 Driver.”

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 using a jdbcKona driver.

If you use the 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, WLE allows you to define a pool of JDBC database connections. You can use the JDBC connection pooling features in WLE CORBA Java and WLE EJB applications.

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

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

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

About the JDBC Drivers and Connection Pooling

The WLE software provides the following JDBC drivers:

- WLE JDBC/XA for Oracle 8.1.5, also referred to as Oracle 8i (Type 2)
- jdbcKona/Oracle 7.3.4 (Type 2)
- jdbcKona/Oracle 8.0.4 (Type 2, for HP-UX systems)

- jdbcKona/MSSQLServer (Type 4)

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.

WLE applications that use the JDBC/XA driver, for local or distributed transactions, must use connection pooling and Oracle 8i.

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

Table 2-1 JDBC Connection Pooling Options and Requirements

JDBC Driver Category	Without JDBC Connection Pooling	With JDBC Connection Pooling
JDBC drivers supporting the JDBC 1.x API. This includes the jdbc/Kona drivers that are included with WLE: ■ jdbcKona/Oracle 7.3.4 ■ jdbcKona/Oracle 8.0.4 (HP-UX) ■ jdbcKona/MSSQLServer4	Obtain JDBC connections from `DriverManager`	Obtain the WLE provided `DataSource (`which wraps around the driver vendor's `DriverManager`) from the WLE JNDI service provider. Then obtain JDBC connections from `DataSource`
JDBC drivers supporting JDBC 2.0 Extension API pertaining to Connection Pooling	Obtain the JDBC driver vendor's `DataSource` from the WLE JNDI service provider. Then obtain JDBC connections from `DataSource`	Obtain the WLE provided DataSource (which wraps around the JDBC driver vendor's `ConnectionPoolDataSource`) from the WLE JNDI service provider. Then obtain JDBC connections from `DataSource`

JDBC Driver Category	Without JDBC Connection Pooling	With JDBC Connection Pooling
WLE JDBC/XA for Oracle 8.1.5 (8i)	Not applicable. (The WLE JDBC/XA for Oracle 8i driver must be used in conjunction with WLE JDBC connection pooling.)	Obtain the WLE provided `DataSource` from the WLE JNDI service provider. Then obtain JDBC connections from `DataSource`

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 Java server 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 WLE 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"
TUXC0NFIG = "/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 BankApp.jar TellerFactory_1 bank_pool"
        RESTART = N

*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 command line options (CLOPT) of the JavaServerXA that operates in the BANK_GROUP1 group of this WLE application.
- 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 Java server defined in the SERVERS section. SRVID is a required parameter. The value is a number from 1 to 30,001. There is no default value.

- Use the `DRIVER` parameter to define the classname for the JDBC driver being used. The values are:
  - `weblogic.jdbc20.oci815.Driver`
  - `weblogic.jdbc20.oci804.Driver` (for HP-UX systems)
  - `weblogic.jdbc20.oci734.Driver`
  - `weblogic.jdbc20.mssqlserver4.Driver`

  The `weblogic.jdbc20.oci815.Driver` value is the **driver name for both**:
  - The WLE 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 WLE JDBC/XA driver is described in Chapter 1, "Using the WLE JDBC/XA Driver." For this driver, you must use connection pooling and therefore must have a `JDBCCONNPOOLS` section in the application's UBBCONFIG file.

  The jdbcKona/Oracle driver is described in Chapter 4, "Using the jdbcKona/Oracle Driver." 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.

  The `weblogic.jdbc.mssqlserver4.Driver` driver is for the jdbcKona/MSSQLServer4 driver that is provided for NT systems. It is not mandatory that you use JDBC connection pooling with this driver. However, if your application on NT uses connection pooling, you must include a `DRIVER` parameter and the `weblogic.jdbc.mssqlserver4.Driver` value 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 are 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 WLE 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 WLE 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 acount that will access the database for this WLE 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 WLE JDBC/XA driver, 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 WAITFORTIMEOUT parameter is not specified, the default for the WAITFORCONN parameter is Y.
- Use the optional WAITFORTIMEOUT 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 WAITFORTIMEOUT 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` 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` generated using `tmunloadcf`, it does not prompt the user to create a password.

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 reports 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 tmadmin printjdbcconnpool Command

```
>printjdbcconnpool
Pool Name   Grp Name   Srv Id Size Max Size Used
----------- ---------- ------ ---- -------- ----
ejbPool     J_SRVGRP      101    1       15    0
Pool2       J_SRVGRP      102    10      30    3
```

The following is the verbose mode output for a single connnection 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. You may want to expand the maximum size of the pool. To do this, you must:

- Shutdown the WLE application with the `tmshutdown` command
- Edit the WLE 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 following related `JDBCCONNPOOLS` parameters: `INITCAPACITY`, `CAPACITYINCR`, `ALLOWSHRINKING`, `SHRINKPERIOD`, and `WAITFORCONN` or `WAITFORTIMEOUT`. 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 WLE software does not support runtime changes to connection pools in running applications.

T_JDBCCONNPOOLS MIB Class

The BEA TUXEDO infrastructure supports WLE 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 WLE application's UBBCONFIG file are stored in the TMIB classes. The properties defined in these classes are read by the WLE 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 WLE 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 documenation.

API Characteristics

The WLE 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 application directly. However, for a JDBC 1.x driver, the connection object returned to the application is implemented by the WLE 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

WLE data sources also implements the following interfaces as an external contract to the WLE local JNDI Service Provider so that the JNDI Service Provider can interact 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 Paclage API for connection pooling, WLE 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 WLE JDBC connection pooling module also support the following interface as an external contract to the pooled connections of the JDBC drivers: javax.sql.ConnectionEventListener.

Obtaining Connections from a WLE Connection Pool

A WLE application performs the following steps to obtain a JDBC connection from the WLE connection pool.

1. Obtaining WLE JNDI implemenation

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

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

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

2. Obtaining the JDBC data source and connection

   Data sources are registered in the JNDI namespace by WLE Java servers. 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 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 WLE 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 WLE 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 data source. Unlike the JDBC 1.0 `DriverManager.getConnection` API, you do not need to supply the user name, 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 specified, an `SQLException` with 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 WLE but not for standard JDBC). If the values match, it behaves the same as the previous method. Otherwise, a `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
```

Application set the log writer for the data source using this API.

WLE connection pool 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 WLE Java application. The following topics are presented:

- 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 WLE Java Application Using the jdbcKona Drivers

API Support

The WLE 5.0 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 WLE. If a WLE application calls a new JDBC 2.0 method that was not in JDBC 1.22, an `SQLException` will be thrown.

The jdbcKona drivers include both Type 2 and Type 4 drivers. The Type 2 drivers (for Oracle) employ client libraries supplied by the database vendors. The Type 4 drivers (for the Microsoft SQL Server) are 100% pure Java; they connect to the database server at the wire level without vendor-supplied client libraries.

Platforms Supported by the jdbcKona Drivers

The following table lists the platforms supported by the jdbcKona drivers.

Note: The Oracle 8.1.5 driver listed in the table 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.

JDBC Driver	Operating System and Version	Java 2 Software Development Kit (SDK)	DBMS	Client Libraries
jdbcKona/ Oracle (Type 2)	Windows NT 4.0 (SP4)	Java 2 SDK 1.2.2	Oracle 7.3.4 Oracle 8.1.5	Oracle 7.3.4 Oracle 8.1.5
	Solaris 2.6 and 7.0	Java 2 SDK 1.2.1, Production release	Oracle 7.3.4 Oracle 8.1.5	Oracle 7.3.4 Oracle 8.1.5
	HP-UX 11.0	Java 2 SDK 1.2.1	Oracle 8.0.4 Oracle 8.1.5	Oracle 8.0.4 Oracle 8.1.5
jdbcKona/ MSSQLServer4 (Type 4)	NT 4.0 with Service Pack 4	Java 2 SDK 1.2.2	Microsoft SQL Server 6.5 (SP3)	Not applicable

Adding the JAR Files to Your CLASSPATH

Be sure to add the WLE JAR files that include the jdbcKona driver classes to your environment.

Note: In %TUXDIR% (NT) or $TUXDIR (UNIX), the /udataobj/java/jdbc/jdbckona.jar file used in previous WLE Java releases has been removed in the latest release. On systems running the WLE 5.0 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 WLE software:

On UNIX Systems:

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

On NT Systems:

```
%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 WLE Java software kit in the $TUXDIR\bin directory.

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

Table 3-1 Updated jdbcKona/Oracle Driver Names

Windows NT/98/95	Solaris	HP-UX
`weblogicoci734.dll`	`libweblogicoci734.so`	`libweblogicoci804.sl`
`weblogicoci815.dll`	`libweblogicoci815.so`	`libweblogicoci815.sl`

The jdbcKona drivers used in previous WLE Java releases are removed in the latest release. The former driver names were:

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

However, accessing the non-XA jdbcKona drivers with the API used in previous WLE releases is still supported. That is, using the `java.sql.DriverManager` API. For example, JDBC applications from WLE 4.2 should be able to use a WLE 5.0 jdbcKona/Oracle driver, provided you change the platform-specific driver class name for the Oracle driver.

The jdbcKona/Oracle driver class names are as follows:

- `weblogic.jdbc20.oci734.Driver`, for NT and Solaris systems
- `weblogic.jdbc20.oci804.Driver`, for HP-UX systems
- `weblogic.jdbc20.oci815.Driver` (non-XA version), for all systems

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 or Microsoft SQL Server)
- 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 is Extended SQL 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();
```

Refer to Chapter 11 of the JDBC 1.2 specification at the Sun Microsystems, Inc. Web site for a description of Extended SQL.

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 `oci804` (HP-UX sysems) or `oci734`, if you are using Oracle 8.0.4 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 6, "jdbcKona Extensions to the JDBC 1.22 API."

Implementing a WLE Java Application Using the jdbcKona Drivers

This section describes the following steps involved in implementing a simple WLE 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 WLE Java applications using the jdbcKona drivers, see Chapter 4, "Using the jdbcKona/Oracle Driver," and Chapter 5, "Using the jdbcKona/ MSSQLServer4 Driver."

Importing Packages

The classes that you import into your WLE Java server 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 WLE 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 runtime. You can even decide after the program is compiled what DBMS to connect to.

Included in the WLE 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 V2 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

In general, to a connect to a DBMS, you need to perform 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 WLE 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`
jdbcKona/ MSSQLServer4	Type 4	`weblogic.jdbc20. mssqlserver4.Driver`	`jdbc:weblogic: mssqlserver4`

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 Driver," and Chapter 5, "Using the jdbcKona/ MSSQLServer4 Driver." For a complete code example, see "Implementing a WLE Java Application Using the jdbcKona Drivers" on page 3-9.

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 WLE 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:

```
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 we 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 Driver."

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.set`*xxx* 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 `set`*xxx*() 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:

```
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 on 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, in a `try {}` block, and 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 WLE 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++) {
  System.out.println("Column Name: "     + md.getColumnName(i));
  System.out.println("Nullable: "       + md.isNullable(i));
  System.out.println("Precision: "      + md.getPrecision(i));
  System.out.println("Scale: "          + md.getScale(i));
  System.out.println("Size: "           + md.getColumnDisplaySize(i));
  System.out.println("Column Type: "    + md.getColumnType(i));
  System.out.println("Column Type Name: "+ md.getColumnTypeName(i));
  System.out.println("");
}
rs.close();
stmt.close();

Statement stmtdrop = conn.createStatement();
try {stmtdrop.execute("drop procedure proc_squareInt");}
catch (SQLException e) {;}
try {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();
      }
      catch (Exception e) {
        // Deal with failures appropriately
      }
      finally {
        try {conn.close();}
        catch (Exception e) {
          // Catch and deal with exception
        }
      }
  }
}
```

4 Using the jdbcKona/Oracle Driver

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

Data Type Mapping

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

Oracle	jdbcKona/Oracle driver
Varchar	String
Number	Tinyint
Number	Smallint
Number	Integer
Number	Long
Number	Float
Number	Numeric
Number	Double

Oracle	jdbcKona/Oracle driver
Long	Longvarchar
RowID	String
Date	Timestamp
Raw	(var)Binary
Long raw	Longvarbinary
Char	(var)Char
Boolean*	Number OR Varchar
MLS label	String

Note that 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).

Note that the `PreparedStatement.setBoolean` method converts a `VARCHAR` type to `"1"` or `"0"` (`string`), and it converts a `NUMBER` type to 1 or 0 (number).

Connecting the jdbcKona/Oracle Driver to an Oracle DBMS

In general, to make a DBMS connection, you perform the following steps.

Note: See the section “Obtaining Connections from a WLE 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, as in the following example for NT and Solaris systems:

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

On an HP-UX system, the example is as follows:

```
Class.forName("weblogic.jdbc20.oci804.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.

Note that 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 `weblogic:jdbc: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");
```

Note: On an HP-UX system, the first line of the previous example would be as follows:

```
Class.forName("weblogic.jdbc20.oci804.Driver").newInstance();
```

In the example, `DEMO` is the V2 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);
```

Note: On an HP-UX system, the `Class.forName` line in the previous example would be as follows:

```
Class.forName("weblogic.jdbc20.oci804.Driver").newInstance();
```

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 classpath examples, the format is:

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

Where `XXX` is the version of the Oracle database: `734` for version 7.3.4, or `804` for version 8.0.4 (HP-UX systems), 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);
```

Note: On an HP-UX system, the `Class.forName` line in the previous example would be as follows:

```
Class.forName("weblogic.jdbc20.oci804.Driver").newInstance();
```

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 804, 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);
```

Note: On an HP-UX system, the `Class.forName` line in the previous example would be as follows:

```
Class.forName("weblogic.jdbc20.oci804.Driver").newInstance();
```

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 `804`, 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.

Column Definition	Returned by `getObject()`
`NUMBER(P <= 9)`	`Integer`
`NUMBER(P <= 18)`	`Long`
`NUMBER(P >= 19)`	`BigDecimal`
`NUMBER(P <=16, S > 0)`	`Double`
`NUMBER(P >= 17, S > 0)`	`BigDecimal`

5 Using the jdbcKona/ MSSQLServer4 Driver

The jdbcKona/MSSQLServer4 is a Type 4, pure-Java, two-tier driver. It requires no client-side libraries because it connects to the database via a proprietary vendor protocol at the wire-format level. Unlike Type 2 JDBC drivers, Type 4 drivers make no native calls, so they can be used in Java applets.

A Type 4 JDBC driver is similar to a Type 2 driver in many other ways. Type 2 and Type 4 drivers are two-tier drivers: each client requires an in-memory copy of the driver to support its connection to the database.

The API reference for JDBC, of which this driver is a fully compliant implementation, is available online in several formats at the Sun Microsystems, Inc. Web site.

Connecting to an SQL Server with the jdbcKona/MSSQLServer4 Driver

To connect to an SQL Server database in a WLE Java server application, perform the following steps.

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

1. Load the jdbcKona/MSSQLServer4 JDBC driver.
2. Request a JDBC connection.

An efficient way to load the JDBC driver is to invoke the `Class.forName().newInstance()` method, specifying the name of the driver class, as in the following example:

```
Class.forName("weblogic.jdbc20.mssqlserver4.Driver").newInstance(
);
```

After loading the JDBC driver, request a JDBC connection by invoking the `DriverManager.getConnection` method. You invoke this method with a connection URL, which, again, specifies the JDBC driver and other connection information.

There are several ways to specify connection information in the `DriverManager.getConnection` method. The following sections describe three methods.

Method 1

The simplest method is to use a connection URL that includes the database name, host name and port number of the database server, and two additional arguments to specify the database user name and password, as in the following example:

```
Class.forName("weblogic.jdbc20.mssqlserver4.Driver").newInstance(
);
```

```
Connection conn =
  DriverManager.getConnection(
              "jdbc:weblogic:mssqlserver4:database@host:port",
              "sa",               // database user name
               "");                // password for database user
```

In this example, `host` is the name or IP number of the computer running SQL Server, and `port` is the port number the SQL Server is listening on.

Method 2

You can set connection information in a `Properties` object and pass this information to the `DriverManager.getConnection` method. The following example specifies the `server`, `user`, and `password` in a `Properties` object:

```
Properties props = new Properties();
props.put("server",     "pubs@myhost:1433");
props.put("user",       "sa");
props.put("password",   "");

Class.forName("weblogic.jdbc20.mssqlserver4.Driver").newInstance(
);
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:mssqlserver4",
                               props);
```

Method 3

You can add connection options to the end of the connection URL, instead of creating a `Properties` object. Separate the URL from the connection options with a question mark (?), and separate options with ampersands (&), as in the following example:

```
Class.forName("weblogic.jdbc20.mssqlserver4.Driver).newInstance()
;
DriverManager.getConnection(
  "jdbc:weblogic:mssqlserver4:database@myhost:myport?user=
   sa&password=");
```

You can use the `Driver.getPropertyInfo` method to find out more about URL options at run time.

Setting Properties for Microsoft SQL Server 7

The jdbcKona/MSSQLServer4 driver recognizes SQL Server 7 automatically. You must set the `sql7` property in the connection URL or in a `Properties` object to `true` to connect to SQL Server 7. For example, the connection URL for an SQL Server 7 connection would be similar to the following:

```
"jdbc:weblogic:mssqlserver4:pubs@myhost:myport?sql7=true"
```

Using the jdbcKona/MSSQLServer4 Driver in Java Development Environments

The jdbcKona/MSSQLServer4 driver has been used successfully in the Java SDK 1.2 for Sun and Windows NT development environment.

JDBC Extensions and Limitations

This section describes the following JDBC extensions and limitations:

- Support for JDBC Extended SQL
- cursorName Method Not Supported
- java.sql.TimeStamp Limitations
- Querying Metadata
- Changing autoCommit Mode
- Statement.executeWriteText() Methods Not Supported

- Sharing a Connection Object in Multithreaded Applications
- EXECUTE Keyword with Stored Procedures

Support for JDBC Extended SQL

The Sun Microsystems, Inc. JDBC specification includes a feature called SQL Extensions, or SQL Escape Syntax. The jdbcKona/MSSQLServer4 driver supports Extended SQL. For information about this feature, see Chapter 3, "Using the jdbcKona Drivers."

cursorName Method Not Supported

The `cursorName` method is not supported, because its definition does not apply to the Microsoft SQL Server.

java.sql.TimeStamp Limitations

The `java.sql.TimeStamp` class in the Java 2 software is limited to dates after 1970. Earlier dates raise an exception. However, if you retrieve dates using the `getString` method, the jdbcKona/MSSQLServer4 driver uses its own date class to overcome the limitation.

Querying Metadata

You can only query metadata for the current database. The metadata methods call the corresponding SQL Server stored procedures, which operate only on the current database. For example, if the current database is master, only the metadata relative to master is available on the connection.

Changing autoCommit Mode

Invoke the `Connection.setAutoCommit` method with a `true` or `false` argument to enable or disable chained transaction mode. When `autoCommit` is `true`, the jdbcKona/MSSQLServer4 driver begins a transaction whenever the previous transaction is committed or rolled back. You must explicitly end your transactions with a `commit` or a `rollback`. If there is an uncommitted transaction when you invoke the `setAutoCommit` method, the driver rolls back the transaction before changing the mode. Be sure to commit any changes before you invoke this method.

Statement.executeWriteText() Methods Not Supported

The jdbcKona Type 2 drivers support an extension that allows you to write text and image data into a row as part of an `SQL INSERT` or `UPDATE` statement without using a text pointer. This extension, `Statement.executeWriteText()`, requires the DB-Library native libraries, and thus is not supported by the jdbcKona/MSSQLServer4 JDBC driver.

To read and write text and image data with streams, you can use the `prepareStatement.setAsciiStream()`, `prepareStatement.setBinaryStream()`, `ResultSet.getAsciiStream()`, and `ResultSet.getBinaryStream()` JDBC methods.

Sharing a Connection Object in Multithreaded Applications

The jdbcKona/MSSQLServer4 driver allows you to write multithreaded applications in which multiple threads can share a single `Connection` object. Each thread can have an active `Statement` object. However, if you invoke the `Statement.cancel` method on one thread, SQL Server may cancel a `Statement` on a different thread. The `Statement` object that is cancelled depends on timing issues in the SQL Server. To avoid this unexpected behavior, we recommend that you get a separate `Connection` object for each thread.

EXECUTE Keyword with Stored Procedures

A Transact-SQL feature allows you to omit the EXECUTE keyword on a stored procedure when the stored procedure is the first command in the batch. However, when a stored procedure has parameters, the jdbcKona/MSSQLServer4 driver adds variable declarations (specific to the JDBC implementation) before the procedure call. Because of this, it is good practice to use the EXECUTE keyword for stored procedures. Note that the JDBC extended SQL stored procedure syntax, which does not include the EXECUTE keyword, is not affected by this issue.

6 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`, `804`, or `815`). The samples in this chapter show `oci734`. For example:

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

However, you would use `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 or Oracle 8.1.5, respectively.

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

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

If this URL changes and you cannot locate this BEA WebLogic JDBC API, please go to http://e-docs.beasys.com/. On that page, click the link for the BEA WebLogic Server (WLS) Documents or Developer Center. On the WLS page, click the API Reference Manual link. The JDBC classdocs are available from this online document.

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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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 `oci804` (HP-UX systems) or `oci815`, if you are using Oracle 8.0.4 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

R

S

T

U

W

X