BEA WebLogic Server

Using the WebLogic JDBC t3 Driver (Deprecated)

BEA WebLogic Server 6.0
Document Date: March 20, 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Using the WebLogic T3 Driver (Deprecated)

Part Number	Document Date	Software Version
N/A	March 20, 2001	BEA WebLogic Server Version 6.0

Contents

About This Document

Audience v
e-docs Web Site v
How to Print the Document vi
Contact Us! vi
Documentation Conventions vii

1. Using the WebLogic JDBC t3 Driver (Deprecated)

T3 Driver Deprecated 2
Overview of JDBC 3
WebLogic JDBC Architecture 3
Using Third-Party JDBC 2.0 Drivers in a Multitier Configuration 4
The WebLogic JDBC API 5
API Reference 5
WebLogic JDBC Objects and Their Classes 5
Other Classes 6
Upgrading to JDK 1.3 6
How to upgrade 7
Implementing WebLogic JDBC 7
Step 1. Importing packages 9
Step 2. Creating the T3Client 9
Step 3. Setting properties for connecting 12
Step 4. Connecting to the DBMS 20
Cached Connections and Connection Pools 20
Using Connection Pools 24
Creating a Startup Connection Pool 25
Creating a Dynamic Connection Pool 29

Obtaining a Connection from a Connection Pool 31
Managing Connection Pools 33
Inserting, Updating, and Deleting Records 40
Creating and Using Stored Procedures and Functions 41
Final Step. Closing the Connection and Disconnecting the T3Client .. 44
Code Summary 45
Other WebLogic JDBC Features 48
Waiting on Oracle Resources 48
Extended SQL 49
Oracle Array Fetches 49
Multibyte Character Set Support 50
About WebLogic JDBC and Oracle NUMBER Columns 50
Implementing with WebLogic JDBC and the JDBC-ODBC Bridge 51
Step 1. Importing packages 52
Step 2. Creating the T3Client 52
Step 3. Connecting 52
Accessing Data 53
Exception Handling 55
Final Step. Disconnecting and Closing Objects 55
Code Summary 55
Using URLs to Set Properties For a JDBC Connection Using the T3 Driver 59
Where URLs are Used 59
How WebLogic URLs are Structured 59
Specifying a Connection with a Properties Object and a URL 59
Specifying a WebLogic JDBC Connection with a Single URL 61
Shortcuts 62
Quoting Metacharacters in a URL 63
Using IDEs and Wizards 64

About This Document

This document describes how use the deprecated JDBC T3 Driver.

- Chapter 1, “Using the WebLogic JDBC t3 Driver (Deprecated).”

Audience

This document is written for application developers who are interested in building applications requiring database access. It is assumed that readers are familiar with SQL, general database concepts, and Java programming.

e-docs Web Site

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

How to Print the Document

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

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

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

Contact Us!

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

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using

- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

Convention	Usage
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n` *`name`* `-a` *`address`* `[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

1 Using the WebLogic JDBC t3 Driver (Deprecated)

This section discusses the deprecated WebLogic JDBC t3 driver in the following topics:

- T3 Driver Deprecated
- Overview of JDBC
- The WebLogic JDBC API
- Implementing WebLogic JDBC
- Other WebLogic JDBC Features
- Implementing with WebLogic JDBC and the JDBC-ODBC Bridge
- Using URLs to Set Properties For a JDBC Connection Using the T3 Driver

T3 Driver Deprecated

The t3 driver is deprecated in WebLogic Server Version 6.0. BEA recommends you use the RMI driver in place of the t3 driver. Refer to Migrating WebLogic Server 4.5 and 5.1 Applications to Version 6.x in the http://e-docs.bea.com/wls/docs60/notes/migrate.html.

Although this document is not being updated, major differences between this release and WebLogic Server Version 5.1 are noted in the following table:

Table 1-1 Resources for Depreciated t3 Driver

Use this feature . . .	To replace . . .	As described here . . .
RMI Driver	t3 Driver	Whenever possible, use the RMI driver in place of the t3 driver. For information on the RMI driver, see Using the WebLogic RMI Driver in *Programming WebLogic JDBC*.
myDriver.connect()	DriverManager.getConnection()	`DriverManager.getConnection()` is a synchronized method, which can cause your application to hang in certain situations. For this reason, BEA recommends that you substitute the **`Driver.connect()`** method for `DriverManager.getConnection()`.
Administration Console	weblogic.properties file	Use the Administration Console to set attributes. This replaces the weblogic.properties file. For more information, see Managing JDBC Connectivity in the *Administration Guide*.
Attributes	weblogic.properties file	To view the list of JDBC attributes, see JDBC Connection Pool in the online help at (DOCROOT}/ConsoleHelp/jdbcconnectionpool.html..

Overview of JDBC

WebLogic JDBC, WebLogic's multitier JDBC implementation (based on JavaSoft's JDBC specification), gives applications database access from within WebLogic. WebLogic supports multiple multitier server products that provide various services, one of which is database access with WebLogic JDBC. Java client applications built with WebLogic JDBC require no client-side database libraries.

WebLogic JDBC requires a JDBC driver between the WebLogic Server and the database server. You can use the WebLogic two-tier driver, WebLogic jDriver for Oracle. You may also elect to use any other JDBC driver, including a JDBC driver for ODBC access. If you choose to use a non-WebLogic JDBC driver between the WebLogic Server and the DBMS, we can only offer you support in deploying WebLogic JDBC after you have demonstrated that your two-tier non-WebLogic JDBC driver works satisfactorily in a two-tier environment.

WebLogic JDBC Architecture

WebLogic JDBC's architecture is defined by its place in the WebLogic framework as part of WebLogic's multitier environment. The WebLogic JDBC Server sits between the WebLogic JDBC application and the remote DBMSes that it accesses. The WebLogic JDBC application is a client of the WebLogic Server, and the WebLogic Server becomes a client of the DBMS. There are two sides to this relationship that are important to understand:

Between the WebLogic Server and its WebLogic JDBC clients. Each WebLogic JDBC client has its own context, or *Workspace*, within WebLogic. The protocol between the WebLogic Server and its clients is a multiplexed, bidirectional, asynchronous connection. The connection between any T3Client and the WebLogic Server is a "rich-socket connection;" that means that the connection carries much more information than a TCP socket connection does. Internally, the WebLogic Server uses an efficient packet-based queuing protocol. For more information about the relationship between T3Clients and the WebLogic Server, see the Developers Guide (deprecated),Writing a T3Client application at http://www.weblogic.com/docs45/classdocs/API_t3.html.

Between the WebLogic Server and the DBMS. Using a JDBC driver and vendor libraries, the WebLogic Server maintains a connection to one or more databases on behalf of the T3Client. The WebLogic Server communicates with remote DBMS via JDBC drivers and, depending upon the JDBC driver, a vendor-specific library or ODBC. We talk about this connection between the WebLogic Server and the DBMS as the "two-tier connection." The WebLogic Server may be connected to more than one database for requests from a single client; and the WebLogic Server may be connected to only one database for requests from multiple clients.

There is also a fundamental relationship between a WebLogic JDBC client and the databases that it accesses, with the WebLogic Server in the middle. This relationship is defined by a JDBC Connection object. There are several ways to create and use JDBC Connections; a WebLogic JDBC client can create a JDBC Connection by setting up certain properties, identifying the right JDBC drivers, and making a connection. A JDBC Connection object can be cached (saved) in the T3Client's Workspace on the WebLogic Server. JDBC Connections can also be created when the WebLogic Server starts up, as a pool of connections available to one or more clients.

The WebLogic Server has many features that can enhance a JDBC application, but all that is required of a WebLogic JDBC application is to connect to a WebLogic Server at the beginning of the session and disconnect at the end. The connection between the WebLogic Server and the DBMS is handled transparently.

Using Third-Party JDBC 2.0 Drivers in a Multitier Configuration

Note: WebLogic Server Version 6.0 requires JDK 1.3

You may use third-party JDBC 2.0 drivers with WebLogic JDBC and WebLogic Server. Such use requires that you run WebLogic Server or WebLogic JDBC under Java 2 (JDK 1.2.x). There are currently limitations regarding the use of Java 2 that you should consider when using JDBC 2.0 drivers. These limitations are discussed on the WebLogic platforms page.

When you use a third-party JDBC 2.0 driver in a multitier configuration, all of the driver's calls and the returned data are transparently passed through the WebLogic multitier driver. This allows you to use any functionality available in that driver in a WebLogic multitier configuration.

To use a JDBC 2.0 driver in a multitier configuration, make the following changes in your code:

1. Change the portion of your Java code where you register the JDBC driver:
2. Change the portion of your Java code that contains the JDBC driver URL:
3. If you are using the CLOB or BLOB datatypes, row caching is not supported. To turn off row caching, set the following connection property in your code:

   ```
   weblogic.t3.cacheRows=0
   ```

4. Re-compile your code.

The WebLogic JDBC API

API Reference

```
Package  java.sql
Package  java.math
Package weblogic.jdbc.common
```

WebLogic provides **extensions to JDBC** for certain WebLogic JDBC enhancements (some support WebLogic jDriver JDBC extensions in a multitier environment). See the API Reference for links to the API (Javadoc) documentation for these extensions.

WebLogic JDBC Objects and Their Classes

Note: For WebLogic Server Version 6.0, JDBC 2.0 is implemented.

The JDBC implementation is not covered in this developer guide. However, we do provide JavaSoft's classdocs (API reference materials) along with our other online reference. Anyone can freely download the JDBC classes and API documentation; go to JavaSoft. Documented here are those objects and classes specific to WebLogic JDBC's use in the WebLogic framework.

The classes that you import in WebLogic JDBC applications include:

- `java.sql.*` for use with the driver `weblogic.jdbc.t3.Driver`
- `weblogic.common.T3Client`
- `java.util.Properties`

Other Classes

`weblogic.common.T3Client`

`weblogic.common.T3User`

`weblogic.common.T3Exception`

The `weblogic.common.*` package contains the T3Client class, which instantiates a T3Client, the T3-specific object that WebLogic JDBC uses to function within the WebLogic framework. Also included in this package is the T3User class. A T3User object is used to pass username and password information to a secure WebLogic Server, that is, one that requires authentication for access.

`weblogic.jdbc.t3.Connection`

WebLogic has extended JDBC for the T3Client to be able to reset the cacheRows property on the connection. Only the extensions are covered in WebLogic's API reference and documentation; other information about JDBC is available at the sun site.

`java.util.Properties`

The java.util.Properties object is used as an argument to construct the JDBC Connection object.

Upgrading to JDK 1.3

With WebLogic release 3.0, you should have upgraded your use of WebLogic JDBC to the latest release of the 1.1 version of the Java Developers Kit WebLogic no longer supports JDK 1.0.2. Running the WebLogic Server against the 1.0.2 JVM is no longer supported, and running or compiling your WebLogic applications against 1.0.2 is also no longer supported.

A significant change between the 1.0.2 and 1.1 versions of the JDK is that the JDBC classes (`java.sql.*`) are included in the JDK 1.1. The temporary set of JDBC classes that WebLogic provided—`xjava.sql.*` and `weblogic.db.xjdbc.*`—are no longer necessary for use with JDK 1.1.

How to upgrade

1. Change the import statements that reference `xjava.sql.*` and `weblogic.db.xjdbc.*` to `java.sql.*` and `weblogic.db.jdbc.*`. Here is an example:

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

Details. Change all references in your code from `xjava.sql.*` to `java.sql.*` when you begin using JDK 1.1. (If you are using dbKona, you will also need to change references to `weblogic.db.xjdbc.*` to `weblogic.db.jdbc.*`. This may mean just changing the import statements. You should also check your code for explicit references to `xjava.sql.*` and `weblogic.db.xjdbc.*` classes.

There are implementation examples in the next section.

2. Change the references to the WebLogic JDBC driver class name to `weblogic.jdbc.t3.Driver`; change the reference to the WebLogic JDBC URL to `jdbc:weblogic:t3`.

Details. One change we have made is the introduction of a new naming convention to make the change to JDK 1.1 more consistent. We have introduced a new class, `weblogic.jdbc.t3.Driver`, which is identical to `weblogic.jdbc.t3client.Driver`, but it uses `java.sql.*` instead of `xjava.sql.*`. You should change references to the WebLogic JDBC driver URL to *jdbc:weblogic:t3* and to the WebLogic JDBC driver class name to *weblogic.jdbc.t3.Driver*.

Implementing WebLogic JDBC

Steps to building a WebLogic JDBC application are covered here. The simple application that is used in the full code example makes a connection to an Oracle database via a WebLogic Server, inserts, updates, and deletes a series of records, and

creates and uses stored procedures and functions. Other examples included in the step-by-step discussion show similar code for use with a Sybase database, particularly in the section on stored procedures and functions.

The first five steps are covered in the order in which they should appear in an application, and are numbered accordingly.

- Step 1. Importing packages
- Step 2. Creating the T3Client
  - Using an explicit or an embedded T3Client
- Step 3. Setting properties for connecting
  - Properties to be set for the two-tier connection
  - Properties to be set for the multitier connection
  - Using a URL to set WebLogic JDBC properties
  - Setting up an embedded T3Client
- Step 4. Connecting to the DBMS
  - Using a named, cached JDBC Connection
  - Creating a startup connection pool
  - Creating a dynamic connection pool
  - Managing connection pools
- Inserting, updating, and deleting records
- Creating and using stored procedures and functions
- Final Step. Disconnecting the T3Client
- Code summary
- Other WebLogic JDBC features
  - Waiting on Oracle resources
  - Extended SQL
  - Oracle array fetches
  - Multibyte character set support

A full code example for using an Oracle database with WebLogic JDBC is reproduced at the end of this document. It incorporates many of the code examples used in the step-by-step instructions.

If you are using a WebLogic jDriver two-tier driver between the WebLogic Server and the DBMS, you should also check the Developers Guides for the particular two-tier driver you are using.

Step 1. Importing packages

Import the following into your WebLogic JDBC application:

- `java.sql.*`
- `weblogic.common.*`
- `java.util.Properties`

You import `java.util.Properties` so that you can create a Properties object to set parameters for accessing the DBMS. The `weblogic.common.*` package contains classes that are shared by all applications that function within the WebLogic framework. For more information on these classes, see WebLogic JDBC and their classes.

Declare the Connection object at the top of your method, since it is used in the `try` and the finally blocks.

Also, declare the T3Client object before the `try` block **unless you will be using an** embedded T3Client. Added as a feature in release 2.3.2, an embedded T3Client is set up by adding another property to the `java.util.Properties` object. An embedded T3Client is constructed and connected automatically, and can be used in any WebLogic JDBC class where you do not need an explicit T3Client object for other operations.

Step 2. Creating the T3Client

Using an explicit or an embedded T3Client

Note: Workspaces are no longer part of WebLogic Server.

In general, each WebLogic JDBC application begins with the creation of a T3Client object. (A T3Client may be created automatically for you, if you use the *embedded T3Client* feature available in release 2.3.2 and after.) The T3Client becomes the client

context for this client within a WebLogic Server and uniquely identifies the client and its requests. A T3Client also owns its Workspace inside the WebLogic Server which persists and allows a T3Client to reconstitute itself for successive WebLogic Server sessions. For WebLogic JDBC applications, the T3Client is also one of the properties passed to the constructor for the JDBC Connection.

The T3Client class, weblogic.common.T3Client, contains constructors and several methods that we use in WebLogic JDBC. You can create a new T3Client, or you can reconstitute a previously created Workspace and return the T3client to the state in which you left it. A client's Workspace includes access to a set of cached JDBC connections that are set aside in the WebLogic Server for a group of WebLogic JDBC Clients.

All T3Client constructors require at least one argument, which is the URL of the WebLogic Server and (optionally, if the port is not 80) the TCP port at which the WebLogic Server is listening for T3Client connection requests. The URL is expressed after the format:

```
accessProtocol://WebLogicServerURL:port
```

The *accessProtocol* may be any of the protocols described in Writing a T3Client application, including **t3** (standard T3Client access over a high-performance, multiplexed, asynchronous, bidirectional connection), **t3s** (T3Client access authenticated and/or encrypted with SSL), or **http** (T3Client access over HTTP tunneling, for transfirewall access).

The *WebLogicServerURL* is determined by whatever method is appropriate on that machine. The *port* is the port at which the WebLogic Server is listening for T3Client login requests.

Here is an example of constructing a T3Client:

```
T3Client t3 = null;
try {
  t3 = new T3Client("t3://bigbox:7001");
```

Each T3Client has a *Workspace* inside the WebLogic Server into which it can save its context and revisit at a later time. You can even name the T3Client's Workspace at the time you create the T3Client, which offers an easy way to revisit the Workspace; or you can get the default T3Client's Workspace after the T3Client connects. Workspaces are powerful models for creating integrated business applications; read more about Workspaces in Using the T3Client Workspace.

For WebLogic JDBC clients, a named Workspace area includes access to any cached JDBC connections to the database (the connectionID), as well as the values set for hard, soft, and idle disconnects. A T3Client can also save arbitrary objects, identified by a String key, into its Workspace for later retrieval.

One way to create a revisitable Workspace is by naming it. You name the Workspace when the T3Client is constructed by providing a String name as an argument to the constructor. For example:

```
t3 = new T3Client("t3://bigbox:7001", "mike");
```

Another way of creating a reenterable Workspace is by saving the String ID of the T3Client's default Workspace, which can be used to reenter it. The ID of the Workspace is available after the T3Client has connected to the WebLogic Server. Here is an example:

```
t3 = new T3Client("t3://bigbox:7001");
t3.connect();
String wsid = t3.services.workspace().getWorkspace().getID();
```

You can then use the Workspace ID (or a T3Client's name) to create a new T3Client, as follows:

```
t3.disconnect();
t3 = null;

// Reconnect using the "wsid"
System.out.println("Reconnecting client " + wsid);
t3 = new T3Client("t3://bigbox:7001", wsid);
t3.connect();
```

If you supply a string that does not match any Workspace IDs or T3Client names currently known to the WebLogic Server, it will assume that you want to create a new Workspace and will name it with the supplied string.

When you have completed work with the WebLogic Server, you should dispose of the client's resources by calling the `disconnect()` method in a finally block (covered in the Final Step below).

You can set several disconnect timeouts for a T3Client, after you have a T3Client object. WebLogic supports timeouts for hard disconnects (where the socket between the WebLogic Server and its client goes away) and soft disconnects (where the T3Client requests a disconnect by calling its `disconnect()` method). You can set timeouts for these (in minutes) to delay cleanup of the T3Client object in the WebLogic Server, as follows:

```
t3.setSoftDisconnectTimeoutMins(5);
```

The default for both hard and soft disconnects is an immediate cleanup of the T3Client resources in the WebLogic Server. You can also set these values to `T3Client.DISCONNECT_TIMEOUT_NEVER` to set the timeout to unlimited.

Setting the disconnect timeout determines how long the WebLogic Server will wait after a particular type of disconnect to clean up (destroy) a T3Client's resources—specifically, its Workspace. Setting a T3Client's soft disconnect timeout to `DISCONNECT_TIMEOUT_NEVER` essentially extends the lifetime of the T3Client for as long as the WebLogic Server runs.

Step 3. Setting properties for connecting

After you have established a WebLogic Server session, you can begin the process of connecting to the DBMS through the WebLogic Server. You will use JDBC Properties objects for setting connection parameters.

The Properties objects contain all the information necessary for making a JDBC Connection. A set of Connections can be cached inside a T3Client's Workspace for that T3Client's use, and the T3Client can request one of its cached connections when it executes the `getConnection()` method with a *connection ID*. Connection IDs are created very simply: by setting a Property. If the connection ID exists, then it is used; otherwise, a new connection ID is created, and with it is saved all of the other Property information for the JDBC Connection. When you used the cached Connection again, all you will need is the connection ID; the Properties objects are ignored if a connection ID exists.

Properties contain the details about how a WebLogic JDBC client should access both the database and the WebLogic Server. After a Properties object is constructed, you can set any number of properties with the `put()` method, by supplying two arguments: the name of the property and its String value. The properties that we set here in the examples are used by the WebLogic JDBC drivers. You may include any other properties that your particular JDBC driver requires.

We use two Properties objects. The first, *dbprops* in the example below, set parameters for the connection between the WebLogic Server and the DBMS, which you can think of as the two-tier connection.

The second Properties object, *t3props* in the example below, sets parameters for the connection between the WebLogic JDBC Client and the DBMS, with the WebLogic Server between them; we also refer to this as the multitier connection. The WebLogic

Server-DBMS Properties object (*dbprops*) will itself be set as a property of the WebLogic JDBCClient-WebLogic Server-DBMS Properties object (*t3props*), which will then be used as an argument for the JDBC Connection constructor.

We use a `java.util.Properties` object to set parameters for the connection between the WebLogic Server and the DBMS (the two-tier connection) and the connection between the WebLogic JDBCClient and the DBMS, with the WebLogic Server between them (the multitier connection). The Properties object is then used as an argument for the `getConnection()` method.

For clarity, we have divided the list of Properties into those that apply to the two-tier connection and those that apply to the multitier connection.

Properties to be set for the two-tier connection

user
: Username for accessing the DBMS.

password
: Password for accessing the DBMS.

server
: Name of the DBMS. The *server* property may also be set as part of the URL in the multitier property *weblogic.t3.driverURL*, added after the URL for the driver, as in "weblogic:jdbc:oracle:DEMO" for an Oracle DBMS with the V2 alias "DEMO".

db or *database*
: Name of the database. Required by some JDBC drivers.

There are additional optional properties that you may set for the two-tier connection, including:

weblogic.oci.cacheRows
: (Oracle only) Support for Oracle array fetches. Calling `ResultSet.next()` the first time will get an array of rows and store it in memory, rather than retrieving a single row. Each subsequent call to `next()` will read a row from the rows in memory until they are exhausted, and only then will `next()` go back to the database.

Note that there is a different property, weblogic.t3.cacheRows, which adjusts the size of the multitier cache for buffering records on the WebLogic Server. These properties are not related although they may be used together.

Properties to be set for the multitier connection

weblogic.t3 **or** *weblogic.t3.serverURL*

Set *weblogic.t3* to the T3Client object. The T3Client has a unique context within the WebLogic Server that is defined and maintained by this object. If you have need for an explicit T3Client object in your program, you will set this property.

If you are using an embedded T3Client, you will **not** set this property, but will set instead the property *weblogic.t3.serverURL*, which identifies the WebLogic Server to which the embedded client will connect.

weblogic.t3.dbprops

The properties object for the two-tier connection itself becomes a property for the multitier connection.

weblogic.t3.driverClassName

Classname of the JDBC driver *between the WebLogic Server and the DBMS*. This may be the class name of any JDBC driver. The string set by this property is used as an argument for the `Class.forName().newInstance()` method on the WebLogic Server. The driver used in this example is the WebLogic jDriver for Oracle native JDBC driver. Note that evaluation of the class name of the driver is case-sensitive. As with all classnames, this class name is in **dot-notation**.

weblogic.t3.driverURL

URL of the two-tier JDBC driver *between the WebLogic Server and the DBMS* (and optionally, the database server name, if not set as the *server* property). This URL is for the driver whose class name is specified in the weblogic.t3.driverClassName property. (More info on this is available in Using URLs with WebLogic products.) The string set by this property is used as the first argument for the `DriverManager.getConnection()` method on the WebLogic Server. In this example, we are accessing a Oracle database "DEMO" with the WebLogic jDriver for Oracle. If you do not supply a server name, the system will look for an environment variable (ORACLE_SID in the case of Oracle). Evaluation of the URL is not case-sensitive. The URL is delimited by colons.

URLs for the WebLogic 2-tier native drivers are:

- jdbc:weblogic:informix4
- jdbc:weblogic:oracle

- jdbc:weblogic.mssqlserver4

weblogic.t3.connectionID (optional)

A named, cached JDBC Connection. You can create a Hashtable of Connection objects inside the Workspace of a T3Client that are available for that T3Client's use and persist in the T3Client's Workspace as long as the T3Client is connected to the WebLogic Server. These cached connections allow the T3Client to resume a JDBC Connection over several sessions. A cached connection ID includes all of the properties initially used to establish the JDBC Connection; consequently, if a connection ID is supplied as an argument to the `getConnection()` method, other properties supplied with java.util.Properties objects are used only if the connection ID doesn't already exist on the WebLogic Server.

weblogic.t3.cacheRows (optional)

Number of rows to be cached. This optional property sets the number of rows cached at the client for each roundtrip between the T3Client and the database. Caching rows can improve performance on the client. This property allows you to fine-tune caching for your application.

When a client calls `ResultSet.next()` for the first time, WebLogic fetches a batch of rows from the DBMS and transmits them to the client JVM in a single response. Subsequent calls to `ResultSet.next()`, retrieve the rows cached in client memory, without calling WebLogic. When the client's cache of rows is exhausted, the next call to `ResultSet.next()` is passed to WebLogic, which again retrieves a batch of rows to send to the client cache.

The number of rows fetched in a batch is configurable via the property weblogic.t3.cacheRows. The default for this property was originally 10 rows; with release version 2.5, the default has been increased to 25. You can set it to a larger or smaller size by specifying this property. To turn off caching, set the property to zero. Then each `next()` or `getXXX()` method call will require a single roundtrip between the database and the T3Client.

There are some combinations of applications and DBMSs that require WebLogic to pass records straight through to the client. For example, when an application is processing a cursor, pre-fetching row data causes a skew between the client's sense of cursor position and that of the DBMS driver. For these applications, setting the weblogic.t3.cacheRows property to zero ('0') provides the needed behavior.

You can control the cacheRows property for each query by resetting this property on the Connection object before retrieving the results of a query into

a ResultSet. Use the WebLogic extensions to JDBC found in weblogic.jdbc.t3.Connection.cacheRows() to get the current value and set a new value. Results will continue to be cached according to the current *cacheRow* setting until it is reset.

Note that if you are using WebLogic jDriver for Oracle, you also have access to Oracle's array fetch functionality through the two-tier property weblogic.oci.cacheRows. (Read more about this feature in the Developers Guide for WebLogic jDriver for Oracle.) This property is independent of *weblogic.t3.cacheRows*, but both may be used together to offset the effects of latency and database load on performance.

weblogic.t3.blobChunkSize (optional, for use with WebLogic jDriver for Oracle)
Defines the buffer size used for streaming blobs between WebLogic and the WebLogic client. This property is used in conjunction with the two-tier Oracle driver properties.

weblogic.t3.name (optional)
The name property allows you to set the name of the connection, which will appear in the Console as the name in the connection's ManagedObject display.

weblogic.t3.description (optional)
The description property allows you to set a short description of the connection, which will appear in the Console as the description in the connection's ManagedObject display.

For example, for a connection that is being used to access customer information, you might add these two properties, as shown here:

```
Properties t3props = new Properties();
t3props.put("weblogic.t3.name",     "CustInfo");
t3props.put("weblogic.t3.description",
            "customer info connection");
```

Setting this information will make the display in the WebLogic Console both more informative and easier to use.

There is one additional property listed here for reference. It is not likely to be used with most of the other properties here, since this property accesses a JDBC Connection from a connection pool. When you set this property, you do not need to set any other properties, since the other attributes for the connection are set when the connection pool is created. This property is usually used in the absence of other properties. It is:

weblogic.t3.connectionPoolID (used with the *weblogic.t3* property only)
: Identifies a pool of JDBC Connections that is created for access by certain T3Users. Check Using Connection Pools for more information and a code example. Using a connection from a pool requires that the connection pool was created (by an entry in the `weblogic.properties` file) when the WebLogic Server was started.

It is important to understand that you will set parameters for **two** connections:

- a two-tier connection between the WebLogic Server and the DBMS
- a multitier connection between the WebLogic JDBC client, the WebLogic Server, and the DBMS

In this example, we first set two-tier connection properties to connect to an Oracle database ("mydb") on the database server ("DEMO") with the WebLogic native JDBC driver for Oracle, WebLogic jDriver for Oracle.

```
Properties dbprops = new Properties();
dbprops.put("user",                    "sa");
dbprops.put("password",                "");
dbprops.put("server",                  "DEMO");
dbprops.put("database",                "mydb");
```

Then we set the multitier properties. Note that one of the multitier properties is the two-tier Properties object. Also note that we supply a connection ID as part of the multitier properties; if this connection ID identifies a Connection (and its set of Properties) that already exists, then the other properties are ignored, since they have been saved with the connection ID.

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.dbprops", dbprops);
t3props.put("weblogic.t3.driverClassName",
            "weblogic.jdbc.oci.Driver");
t3props.put("weblogic.t3.driverURL",
            "jdbc:weblogic:oracle");
t3props.put("weblogic.t3.connectionID",
            dbconnid);
t3props.put("weblogic.t3.cacheRows", "100");
t3props.put("weblogic.t3.name", "CustInfo");
t3props.put("weblogic.t3.description",
            "customer info connection");
```

Note that the formats of the class name of the driver and the URL are different; the class name uses dot-notation, and the URL uses separating colons.

Using a URL to set WebLogic JDBC properties

Some development environments set restrictions on the use of a `java.util.Properties` object for setting multiple database properties. For example, Powersoft's PowerJ uses the Properties object exclusively for setting the username and password for the DBMS. WebLogic has developed a URL scheme to supply all of the other information needed for a WebLogic JDBC connection. For more information on setting WebLogic JDBC properties for connection with a URL, Using URLs to Set Properties For a JDBC Connection Using the T3 Driver.

Setting up an embedded T3Client

If you do not need an explicit T3Client object for other purposes in your WebLogic JDBC program, you can use an embedded T3Client. An embedded T3Client is created, connected, and disconnected automatically for you underneath. All you must do is set another property in your `java.util.Properties` object that supplies the URL of the WebLogic Server.

Here is a simple example of using an embedded T3Client; this example uses the `weblogic.jdbc.t3.Driver` for use with JDK 1.1, which contains all of the `java.sql` JDBC classes. Note that there is no declaration nor construction of a T3Client object in this example.

```
  Class.forName("weblogic.jdbc.t3.Driver").newInstance();
  // Set up properties for connecting to the DBMS
  Properties dbprops = new Properties();
  dbprops.put("user",                 "scott");
  dbprops.put("password",             "tiger");
  dbprops.put("server",               "DEMO20");

  Properties t3props = new Properties();
  t3props.put("weblogic.t3.dbprops",         dbprops);
  // Set the URL of WebLogic to create an embedded T3Client
  t3props.put("weblogic.t3.serverURL",     "t3://localhost:7001");
  t3props.put("weblogic.t3.driverClassName",
              "weblogic.jdbc.oci.Driver");
  t3props.put("weblogic.t3.driverURL",
"jdbc:weblogic:oracle");
  t3props.put("weblogic.t3.cacheRows",      "10");

  Connection conn =
    DriverManager.getConnection("jdbc:weblogic:t3", t3props);
  Statement stmt = conn.createStatement();
  stmt.execute("select * from empdemo");
```

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

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

  ResultSetMetaData rsmd = rs.getMetaData();

  stmt.close();
  conn.close();
}
```

You can also set up an embedded T3Client to use a JDBC Connection from a WebLogic JDBC connection pool, by setting the property *weblogic.t3.connectionPoolID*. If you are using a JDBC Connection from a connection pool, you will need only the properties that are required for requesting a connection from the pool, as well as the property *weblogic.t3.serverURL* property. For more information on pools, read Using connection pools in this document.

You can also use an embedded T3Client even if you need to set a username and password for a T3User for T3Client-to-WebLogic security. Here is an example. This sets up access to a connection pool “eng” for which the T3User “development” has been added to the `weblogic.properties` file, as well as a Permission to “reserve" a connection from this pool, with these properties:

```
  weblogic.password.development=3Y(sf40!VmoN

weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=developme
nt
```

Here is how you would use a connection from this connection pool with an embedded T3Client:

```
  Properties t3props = new Properties();
  t3props.put("weblogic.t3.serverURL",
"t3://localhost:7001");
  t3props.put("weblogic.t3.connectionPoolID", "eng");
  t3props.put("weblogic.t3.user",             "development");
  t3props.put("weblogic.t3.password",         "3Y(sf40!VmoN");
```

Note that all of the other properties for connecting, like the location of the database server, are set by the configuration entry in the weblogic.properties file that creates the connection pool at startup. You can find an example entry for a connection pool (commented-out) in the properties file that is shipped with the distribution.

Step 4. Connecting to the DBMS

The WebLogic JDBC client never connects directly to the database, but connects to the WebLogic Server, which accesses the database on behalf of the T3Client. You must supply JDBC drivers for both the two-tier connection (between the WebLogic Server and the DBMS) and the multitier connection (between the WebLogic JDBC client, the WebLogic Server, and the DBMS). The class name and the URL of the two-tier connection is set with a Properties object, as described above.

For the connection *between the WebLogic JDBC client, the WebLogic Server, and the DBMS*, you will supply the class name and URL for the WebLogic JDBC driver, which manages the multitier connection.

The WebLogic JDBC driver class name is supplied by calling the `Class.forName().newInstance()` method with the class name **weblogic.jdbc.t3.Driver**. Calling `Class.forName().newInstance()` properly loads and registers the driver class.

You supply the URL **jdbc:weblogic:t3** for this driver as an argument to the `DriverManager.getConnection()` method.

In this example, we use the `Class.forName().newInstance()` method to identify and load the WebLogic JDBC driver. Then we create the JDBC Connection with the URL of the WebLogic JDBC driver and a Properties object:

```
// Class name for the WebLogic JDBC
Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3",
                              t3props);
```

Once you have established a Connection (that is, constructed a JDBCConnection object), you use WebLogic JDBC methods just as you would anyother implementation of JDBC.

Cached Connections and Connection Pools

WebLogic provides ways to supply your T3Clients with reusable JDBC Connections. Logging into a DBMS can be an expensive and time-consuming operation. With a reusable connection, the overhead of connecting to the DBMS is incurred just once, when the connection is created. The WebLogic Server holds the connection open until it is needed again.

One type of reusable connection is a named, cached JDBC Connection. Another type is a connection pool.

A cached JDBC Connection is created, probably used, and then saved in the T3Client's Workspace for later use. Caching a JDBC Connection saves overhead in having to create a Connection over and over again, but a cached Connection also ties up database resources on a long-term basis. When you cache a JDBC Connection, you save its entire state with it.

A pool of JDBC Connections can be created when the WebLogic Server starts up, before there are any requests for JDBC Connections, or dynamically by a T3Client. You can also create a connection pool dynamically using the `weblogic.Admin` class create_pool command.

As T3Clients connect to the WebLogic Server, they can obtain a connection from the pool, and then return it to the pool when finished. Creating a pool of connections is a good way to allocate scarce resources (like database connections) among multiple clients. You can set the pool to grow incrementally when all of the connections have been allocated, up to a maximum number of connections. You can also assign sets of users to named pools. A connection pool saves WebLogic Server and DBMS overhead in creating connections, since the connections are created once, and then reused over and over again.

Connection pools and cached connections are different. A cached JDBC Connection is created by a particular T3Client, and the details about that JDBC Connection are stored in that T3Client's Workspace. The cached connection lasts only the lifetime of the T3Client. As soon as the WebLogic Server cleans up the resources for that T3Client, the cached connection is destroyed.

Connection pools, on the other hand, are available to any T3Client. The lifetime of a connection from a connection pool is not tied in any way to the lifetime of a T3Client. When a T3Client closes a connection from a connection pool, the connection is returned to the pool and becomes available again for other T3Clients, but the connection itself is not closed.

Following are more details on how to create and use cached JDBC Connections and connection pools. These are specifically "client-side" Connections and connection pools. You can also take advantage of another WebLogic enhancement to JDBC for *server-side* connection pools, for use in HTTP servlets and other applications that do not use a T3Client. For more information on using server-side connection pools, read the pertinent section in the Developers Guide, Using WebLogic HTTP servlets.

Using a Named, Cached JDBC Connection

You can create and name a JDBC Connection and cache the Connection on the WebLogic Server in the T3Client's Workspace so that you can reuse the same JDBC Connection again and again. A T3Client's cached JDBC Connection will last the lifetime of the T3Client's persistence on the WebLogic Server. You can make this lifetime indefinite—thus giving the T3Client a permanent preserved Workspace on the WebLogic Server that will last the lifetime of the WebLogic Server—by setting the T3Client's soft disconnect timeout to DISCONNECT_TIMEOUT_NEVER. Once you have set the soft disconnect timeout to never, disconnecting the T3Client by calling the `T3Client.disconnect()` method will not cause the WebLogic Server to reclaim the T3Client's resources.

In this class, we create a named JDBC Connection that we reuse several times. We identify the named JDBC Connection with the `java.util.Property`*connectionID*.

This class has two methods, a static `getConnection()` method to create and then reuse the JDBC Connection, and a `main()`. First, let's examine the code for the `getConnection()` method in this class.

This method returns a JDBC Connection object; if we supply a property "connectionID" that already exists on the WebLogic Server, all of the other properties for login access, etc., are ignored, and the cached JDBC Connection, which contains all the information necessary to resume connection to the database, is used instead.

```
static Connection getConnection(T3Client t3, String dbconnid)
  throws Exception
{

  // If a connectionID is given, the other properties are used
 // only if the connectionID doesn't exist on the WebLogic Server.
  // Other values are ignored if the connectionID exists.
  Properties dbprops = new Properties();

  // Set the two-tier props
  dbprops.put("user",                    "scott");
  dbprops.put("password",                "tiger");
  dbprops.put("server",                  "DEMO");

  Properties t3props = new Properties();
  // Set the multitier props, including the connectionID
  t3props.put("weblogic.t3",             t3);
  t3props.put("weblogic.t3.dbprops",     dbprops);
  t3props.put("weblogic.t3.driverClassName",
              "weblogic.jdbc.oci.Driver");
  t3props.put("weblogic.t3.driverURL",
```

```
                "jdbc:weblogic:oracle");
    // If dbconnid has been cached, all the preceding properties
    // are ignored. The connection is instant
    t3props.put("weblogic.t3.connectionID",   dbconnid);

    Class.forName("weblogic.jdbc.t3.Driver").newInstance();
    return  DriverManager.getConnection("jdbc:weblogic:t3",
                                        t3props);
  }
```

In the next code example, we carry out the following steps:

1. Create a T3Client.
2. Connect to the WebLogic Server.
3. Set the soft disconnect timeout for this client to be indefinite.
4. Save the ID of the Workspace so that the T3Client can come back to this Workspace at a later time. You can also supply a name (like "mike") for a client as an argument when you new the T3Client object.

```
  T3Client t3 = new T3Client("t3://localhost:7001");
  t3.connect();

t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);
  String wsid = t3.services.workspace().getWorkspace().getID();
```

Then we create a named JDBC Connection to an Oracle database by calling the `getConnection()` method with the name “myconn.”

```
  System.out.println("Logging into database and " +
                     "saving session as myconn");
  Connection conn = getConnection(t3, "myconn");
```

Now we can disconnect the T3Client and set it to null. Since the soft disconnect timeout is never, the WebLogic Server will preserve this client’s resources, including its Workspace and its named JDBC Connections.

```
  t3.disconnect();
  t3 = null;
```

Next we reconnect to the WebLogic Server by creating a new T3Client named with the Workspace ID of the first client. This relinks the T3Client to the Workspace that we created with the first client; in that Workspace is stored the JDBC Connection that we cached under the name "myconn.”

Then we resume the connection to the Oracle database by calling the `getConnection()` method with the name "myconn." All of the parameters for connecting already exist. We do some arbitrary database work and then close the JDBC Connection object.

```
t3 = new T3Client("t3://localhost:7001", wsid);
t3.connect();

for (int j = 0; j < 5; j++) {
  System.out.println("Reestablishing database connection");
  conn = getConnection(t3, "myconn");

  System.out.println("Performing query");
  QueryDataSet qds = new QueryDataSet(conn, "select * from emp");
  qds.fetchRecords();
  System.out.println("Record count = " + qds.size());
  qds.close();
}
conn.close();
```

Once you have finished with a T3Client's resources, you can instruct the WebLogic Server to reclaim those resources by setting the T3Client's soft disconnect timeout to zero, which effects a cleanup as soon as the T3Client calls the `disconnect()` method, as shown here:

```
t3.setSoftDisconnectTimeoutMins(0);
  t3.disconnect();
```

Using Connection Pools

Another method of connecting is by creating a pool of JDBC Connections from which a T3User can request a connection. You can define a connection pool in the `weblogic.properties` file, called a "startup" connection pool, or you can create a "dynamic" connection pool in a running WebLogic Server from within a T3Client application.

Creating a pool of JDBC Connections gives T3Clients ready access to connections that are already open. It removes the overhead of opening a new connection for each DBMS user, since the connections in the pool are shared among the members of the group.

Creating a Startup Connection Pool

A startup connection pool is declared in the `weblogic.properties` file. The WebLogic Server opens JDBC connections to the database during the WebLogic startup process and adds the connections to the pool.

You define a startup connection pool with an entry after the following pattern in your `weblogic.properties` file. An example is commented out in the properties file that is shipped with the distribution, under the heading "JDBC Connection Pool Management:"

```
weblogic.jdbc.connectionPool.VirtualName=\
         url=JDBC driver URL,\
         driver=full package name for JDBC driver,\
         loginDelaySecs=seconds between each login attempt,\
        initialCapacity=initial number of connections in the pool,\
         maxCapacity=max number of connections in the pool,\
        capacityIncrement=number of connections to add at a time,\
         allowShrinking=true to allow shrinking,\
         shrinkPeriodMins=interval before shrinking,\
         testTable=name of table for connection test,\
         refreshTestMinutes=interval for connection test,\
         testConnsOnReserve=true to test connection at reserve,\
 testConnsOnRelease=true to test connection at release,\
         props=DBMS connection properties

weblogic.allow.reserve.weblogic.jdbc.connectionPool.name=\
         T3Users who can use this pool
weblogic.allow.reset.weblogic.jdbc.connectionPool.name=\
         T3Users who can reset this pool
weblogic.allow.shrink.weblogic.jdbc.connectionPool.name=\
         T3Users who can shrink this pool
```

The information that you supply is shown above in red. Required information is noted. If you do not supply a value that is required, an exception is thrown when you start the WebLogic Server.

Here is a short description of the arguments for this property:

name

(Required) Name of the connection pool. You will use the name to access a JDBC Connection from this pool when you write your T3Client class.

url

(Required) URL of the JDBC 2-tier driver for the connection between the WebLogic Server and the DBMS. You can use one of the WebLogic jDrivers

or another JDBC driver that you have tested in a 2-tier environment. Check the documentation for the JDBC driver you choose to find the URL.

driver

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

loginDelaySecs

(Optional) Number of seconds to wait between each attempt to open a connection to the database. Some database servers can't handle multiple requests for connections in rapid succession. This property allows you to build in a small delay to let the database server catch up.

initialCapacity

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

maxCapacity

(Required) The maximum size of the pool.

capacityIncrement

(Required) The size by which the pool's capacity is enlarged. *initialCapacity* and *capacityIncrement* work somewhat like a Java Vector, which has an initial allocation (its "capacity") and is increased in increments as necessary (capacityIncrement), up to the pool *maxCapacity*.

allowShrinking

(Optional. Introduced in 3.1) Whether this connection pool should be allowed to shrink back to its initial capacity, after expanding to meet increased demand. Set *shrinkPeriodMins* if this property is set to true, or it will default to 15 minutes. Note that allowShrinking is set by default to false, for backwards compatibility.

shrinkPeriodMins

(Optional. Introduced in 3.1) The number of minutes to wait before shrinking a connection pool that has incrementally increased to meet demand. You must set *allowShrinking* to true to use this property. The default shrink period is 15 minutes and the minimum is 1 minute.

testTable

(Required only if you set refreshTestMinutes, testConnsOnReserve, or testConnsOnRelease. Introduced in 4.0) The name of a table in the database

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

refreshTestMinutes

(Optional, introduced in 4.0) This property, together with the *testTable* property, enables autorefresh of connections in the pools. At a specified interval, each *unused* connection in the connection pool is tested by executing a simple SQL query on the connection. If the test fails, the connection's resources are dropped and a new connection is created to replace the failed connection.

To enable autorefresh, set *refreshTestMinutes* to the number of minutes between connection test cycles—a value greater than or equal to 1. If you set an invalid *refreshTestMinutes* value, the value defaults to 5 minutes. Set *testTable* to the name of an existing database table to use for the test. Both properties must be set to enable the autorefresh feature.

testConnsOnReserve

(Optional, introduced in 4.0.1) When set to true, the WebLogic Server tests a connection after removing it from the pool and before giving it to the client. The test adds a small delay in serving the client's request for a connection from the pool, but ensures that the client receives a working connection. The testTable parameter must be set to use this feature.

testConnsOnRelease

(Optional, introduced in 4.0.1) When set to true, the WebLogic Server tests a connection before returning it to the connection pool. If all connections in the pool are already in use and a client is waiting for a connection, the client's wait will be slightly longer while the connection is tested. The testTable parameter must be set to use this feature.

props

(Required) The properties for connecting to the database, such as **username, password, and server**. The properties are defined by, and processed by, the 2-tier JDBC driver that you use. Check the documentation for the JDBC driver to find the properties required to connect to your DBMS.

allow

This attribute was deprecated in 3.0. Set up access to a connection pool using "reserve" and "reset" Permissions as shown above.

This example, taken from the `weblogic.properties` file that is shipped with the WebLogic distribution, creates a connection pool named "eng," which is accessible to 3 T3Users (Guest, Joe, and Jill). It allocates a minimum of 4 and a maximum of 10 JDBC connections for an Oracle database with a username of "SCOTT," password "tiger," and server name "DEMO." The WebLogic Server sleeps for 1 second between each connection attempt to prevent refused logins from a DBMS that may be under load or on a saturated network. The connection pool shrinks back to 4 connections when connections in the pool are unused for 15 minutes or more. Every 10 minutes, unused connections are tested and refreshed if they have have gone stale.

```
weblogic.jdbc.connectionPool.eng=\
       url=jdbc:weblogic:oracle,\
       driver=weblogic.jdbc.oci.Driver,\
       loginDelaySecs=1,\
       initialCapacity=4,\
       maxCapacity=10,\
       capacityIncrement=2,\
       allowShrinking=true,\
       shrinkPeriodMins=15,\
       refreshTestMinutes=10,\
       testTable=dual,\
       props=user=SCOTT;password=tiger;server=DEMO

weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=\
       guest,joe,jill
weblogic.allow.reset.weblogic.jdbc.connectionPool.eng=\
       joe,jill
weblogic.allow.shrink.weblogic.jdbc.connectionPool.eng=\
       joe,jill
```

Note that if you have a username with a null password, you shouldn't enter an empty string for the password in the connection pool registration; rather you should simply leave it blank. Here is an example taken from the WebLogic Administrators Guide document on properties:

```
weblogic.jdbc.connectionPool.eng=\
       url=jdbc:weblogic:oracle,\
       driver=weblogic.jdbc.oci.Driver,\
       loginDelaySecs=1,\
       initialCapacity=4,\
       capacityIncrement=2,\
       maxCapacity=10,\
```

```
       props=user=sa;password=;server=demo
weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=guest,joe
,jill
```

Creating a Dynamic Connection Pool

A JNDI-based API introduced in WebLogic release 4.0 allows you to create a connection pool from within a T3Client application. With this API, you can create a connection pool in a WebLogic Server that is already running.

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

A property in the `weblogic.properties` file, `weblogic.allow.admin.weblogic.jdbc.connectionPoolcreate`, determines who can create dynamic connection pools. If the property is not set, then only the "system" user can create a dynamic connection pool.

For example, the following property allows users "joe" and "jane" to create dynamic connection pools:

```
weblogic.allow.admin.weblogic.jdbc.connectionPoolcreate=joe,jane
```

You can also create ACLs for dynamic connection pools by adding `weblogic.allow.reserve.`*ACLname* and `weblogic.allow.admin.`*ACLname* entries to the `weblogic.properties` file. For example, the following two properties define an ACL named "dynapool" that allows anyone (the "everyone" group) to use a connection pool, and users "joe" and "jane" to administer a connection pool:

```
weblogic.allow.admin.dynapool=joe,jane
weblogic.allow.reserve.dynapool=everyone
```

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

To create a dynamic connection pool in a T3 application, you get an initial JNDI context to the WebLogic JNDI provider, and then look up "weblogic.jdbc.common.JdbcServices." This example shows how this is done:

```
Hashtable env = new Hashtable();

env.put(java.naming.factory.initial,
                "weblogic.jndi.WLInitialContextFactory");
// URL for the WebLogic Server
env.put(java.naming.provider.url, "t3://localhost:7001");
env.put(java.naming.security.credentials,
         new T3User("joe", "joez_secret_wrdz"));

Context ctx = new InitialContext(env);

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

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

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

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

try {
  // Set properties for the Connection Pool.
  // The properties are the same as those used to define a startup
  // connection pool in the weblogic.properties file.
  Properties poolProps = new Properties();

  poolProps.put("poolName",        "eng2");
  poolProps.put("url",             "jdbc:weblogic:oracle");
  poolProps.put("driver",          "weblogic.jdbc.oci.Driver");
  poolProps.put("initialCapacity", "1");
  poolProps.put("maxCapacity",     "10");
  poolProps.put("props",           "user=SCOTT;
                                   password=tiger;server=DEMO");
  poolProps.put("aclName",        "dynapool");  // the ACL to use

  // Creation fails if there is an existing pool with the same
```

```
name.
    jdbc.createPool(poolProps);
  }
  catch (Exception e) {
     system.out.Println("Error creating connection pool eng2.");
  }
  finally { // close the JNDI context
         ctx.close();
  }
```

Obtaining a Connection from a Connection Pool

Using a connection from a connection pool is nearly the same as opening a JDBC connection. The JDBC driver class is weblogic.jdbc.t3Client.Driver and the connection URL is "jdbc:weblogic:t3". To identify the connection pool you want to use, you create a java.util.Properties object and set a property called `weblogic.t3.connectionPoolID` to the name of the connection pool.

Using a connection from a connection pool. You use a connection from a connection pool in your client application by creating a java.util.Properties object, and setting a property called `weblogic.t3.connectionPoolID` to the name of the connection pool you created in the WebLogic Server's `weblogic.properties` file.

In this simple example, we create a T3Client, set up a Properties object, and then open a connection from the connection pool "eng," for which the `weblogic.properties` file entry is shown above.

```
T3Client t3 = new T3Client("t3://bigbox:7001");
  t3.connect();

  // Note that we only need to set two properties,
  // the T3Client and the connectionPoolID
  Properties t3props = new Properties();
  t3props.put("weblogic.t3", t3);
  t3props.put("weblogic.t3.connectionPoolID", "eng");

  Class.forName("weblogic.jdbc.t3.Driver").newInstance();
  Connection conn =
    DriverManager.getConnection("jdbc:weblogic:t3",
                                t3props);
  // Do any arbitrary database work
  QueryDataSet qds = new QueryDataSet(conn, "select * from emp");
  qds.fetchRecords();
  System.out.println("Record count = " + qds.size());
  qds.close();
```

```
    // Release the connection
    conn.close();
    // Disconnect the client
    t3.disconnect();
  }
```

Note that the JDBC Connection is released by this client and returned to the pool before the client disconnects. At the time when the JDBC connection is returned to the connection pool, any outstanding JDBC transactions are rolled back and closed.

Waiting on a pool connection to become available. If all of the connections in a pool are in use, the client will, by default, wait until a connection becomes available. You can change this behavior for a client in two ways:

- Disable the wait. If no connection is available, `DriverManager.getConnection()` gets an immediate exception.
- Specify the number of seconds to wait for a connection. If no connection is available within the number of seconds you specify, `DriverManager.getConnection()` gets an exception.

To disable the wait, set the `weblogic.t3.waitForConnection` property to "false" in the Properties object you pass to `DriverManager.getConnection()`:

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");
t3props.put("weblogic.t3.waitForConnection", "false");

Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:t3",
                               t3props);
```

With the wait disabled, the `DriverManager.getConnection()` call throws an exception immediately if no connection is available.

To specify a period of time to wait for a connection to become available, set the `weblogic.t3.waitSecondsForConnection` property to the number of seconds you want to wait. This example waits for up to 15 seconds:

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");
t3props.put("weblogic.t3.waitSecondsForConnection", "15");
```

```
Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:t3",
                               t3props);
```

Managing Connection Pools

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

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

Retrieving Information About a Pooll

weblogic.jdbc.common.JdbcServices.poolExists

weblogic.jdbc.common.Pool.getProperties

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

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

Disabling a Connection Pool

weblogic.jdbc.common.Pool.disableDroppingUsers

weblogic.jdbc.common.Pool.disableFreezingUsers

weblogic.jdbc.common.pool.enable

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

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

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

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

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

Shrinking a Connection Pool

weblogic.jdbc.common.Pool.shrinking

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

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

Shutting Down a Connection Pool

weblogic.jdbc.common.Pool.shutdownSoft

weblogic.jdbc.common.Pool.shutdownHard

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

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

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

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

Resetting a Pool

weblogic.jdbc.common.Pool.reset

weblogic.jdbc.t3.Connection

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

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

To refresh a single connection, use the `refresh()` method in the weblogic.jdbc.t3.Connection class. When you call this method, you lose any Statements and Resultsets you had on the connection, plus your application incurs the

relatively high cost of opening the connection. Therefore, we recommend that you only refresh a single connection when you get an error that implies that the connection has gone bad.

You will need to explicitly cast the JDBC Connection as a (`weblogic.jdbc.t3.Connection`). Other than that, the `refresh()` method in the Connection class is used in the same way the `reset()` method is used for a connection pool. First try an execute action on the Connection that is guaranteed to succeed if the Connection itself is viable. Catch the exception and call the `refresh()` method.

Here is an example. Notice that we cast the JDBC Connection as a `weblogic.jdbc.t3.Connection` in the last line of this example, in order to take advantage of the WebLogic extension to JDBC.

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();

Properties t3props = new Properties();
t3props.put("weblogic.t3",                   t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");

Class.forName("weblogic.jdbc.t3c.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3client",
                              t3props);

try {
  Statement stmt = conn.createStatement();

  // This SQL is guaranteed to succeed over a good connection
  // to an Oracle DBMS
  ResultSet rs = stmt.executeQuery("select 1 from dual");

  if (rs != null) {
    while (rs.next()) {;}
  }

  rs.close();
  stmt.close();
  conn.close();
}

catch(SQLException e) {

  // We cast the JDBC Connection as a
  // weblogic.jdbc.t3.Connection
```

```
    // and call the refresh() method on it
    ((weblogic.jdbc.t3.Connection)conn).refresh();
  }
```

Refreshing a Single Pool Connection

weblogic.jdbc.common.JdbcServicesDef

You can refresh a single connection from a connection pool, or reset the entire connection pool, if one or more connections in the pool go stale. For example, if the DBMS is taken down while the WebLogic Server is actively supporting a pool of connections. A connection pool autorefresh feature, introduced with WebLogic Server 4.0, can also be enabled to periodically test and refresh connections.

There are only certain instances when resetting a pool is appropriate. You should never use this feature as a routine part of a user program. Usually, what makes resetting a connection pool necessary is that the DBMS has gone down and the connections in the pool are no longer viable. Attempting to reset the pool before you are certain that the DBMS is up and available again will cause an Exception to be thrown. Resetting a pool should always be a special operation that is carried out by a user with administrative privileges.

There are several ways to reset connections pools:

- You can use the weblogic.Admin command (as a user with administrative privileges) to reset a connection pool, as an administrator. Here is the pattern:

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

You might use this method from the command line on an infrequent basis. There are more efficient programmatic ways that are also discussed here. For more on the Admin commands, read the WebLogic Administrators Guide, Running and maintaining the WebLogic Server.

- You can use an WebLogic Events class, ActionRefreshPool, to periodically check the viability of your connection pool and automatically refresh it when needed. The ActionRefreshPool is part of WebLogic's public API. Running this class also requires administrative privileges for the WebLogic Server. You can even register this class as a startup class. **You will probably find it easiest to use this method to ensure the viability of connections in your connection pools**.

- You can use the `reset()` method from the jdbcServicesDef interface in your client application.

The last case requires the most work for you, but also gives you more flexibility than the first two. We have provided some sample code here to show you how to use the `reset()` method.

Here is an example of resetting a pool using the `reset()` method.

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

This simple code example that tests the demo JDBC connection pool "eng,” which is a pool of 5 connections to an Oracle DBMS:

```
  String poolID = "eng";

  T3Client t3 = new T3Client("t3://localhost:7001");
  t3.connect();

  Properties t3props = new Properties();
  t3props.put("weblogic.t3",                    t3);
  t3props.put("weblogic.t3.connectionPoolID", poolID);

  Class.forName("weblogic.jdbc.t3.Driver").newInstance();
  Connection conn =
    DriverManager.getConnection("jdbc:weblogic:t3client",
                                t3props);

  try {

   // If the connection isn't operable, you will get an SQLException
if
    // you attempt to execute anything over it
    Statement stmt = conn.createStatement();

    // This SQL is guaranteed to succeed over a good connection
    // to an Oracle DBMS
    ResultSet rs = stmt.executeQuery("select 1 from dual");
```

```
    // Do some arbitrary database work
    if (rs != null) {
      while (rs.next()) {;}
    }

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

    // Release the connection and return it to the pool
    conn.close();
  }

  // If the try block fails, reset the pool. Note that a program
  // that calls this method should have administrator privileges.
  // You should not try to reset the pool until you are certain
  // that the database is available again.

  catch(SQLException e) {
    t3.services.jdbc().resetPool(poolID);
  }
```

Setting up ACLs for Connection Pools in the WebLogic Realm

`weblogic.jdbc.connectionPool`

`weblogic.jdbc.connectionPool.`*poolID*

WebLogic controls access to internal resources like JDBC connection pools through ACLs set up in the WebLogic Realm. Entries for ACLs in the WebLogic Realm are listed as properties in the `weblogic.properties` file.

You can set the Permissions "reserve," "reset," and "shrink" for JDBC connections in a connection pool by entering a property in the properties file. Setting a Permission for the ACL "weblogic.jdbc.connectionPool" limits access to all connection pools. Add Permissions for other users by adding an entry for the ACL name "weblogic.jdbc.connectionPool.poolID," which controls access to the connection pool *poolID*. The special user *system* always has Permissions "reserve," "reset," and "shrink" for every ACL, no matter what other Permissions have been set.

Example:

```
weblogic.allow.reserve.weblogic.jdbc.connectionPool.\
eng=margaret,joe,maryweblogic.allow.reset.weblogic.jdbc.\
connectionPool.eng=sysMonitorweblogic.allow.shrink.\
weblogic.jdbc.connectionPool.eng=sysMonitor
```

For backwards compatibility, you can also use the old-style property syntax to grant permission for "reserve" by setting a userlist for the property weblogic.jdbc.connectionPool.*poolID*=allow=. It is recommended that you upgrade your properties file as soon as possible to reflect the new usage, since WebLogic cannot guarantee how long it will support old-style properties.

Inserting, Updating, and Deleting Records

You use a JDBC Statement or one of its subclasses, created in the context of the JDBC Connection, to execute queries on the database. The results of the query are contained in a JDBC ResultSet. Use the `next()` and `getXXX()` methods in the ResultSet class to access rows in the database.

In this example, we first insert ten records into the Employee table. We use a PreparedStatement with the JDBC PreparedStatement "?" syntax. We can make updates to the records using the `setInt()` and `setString()` methods from the PreparedStatement class.

```
String inssql = "insert into emp (empno, empname) values (?, ?)";
 PreparedStatement pstmt = conn.prepareStatement(inssql);
 for (int j = 0; j < 10; j++) {
   pstmt.setInt(1, i);
   pstmt.setString(2, "John Smith");
   pstmt.execute();
 }
 pstmt.close();
```

After inserting the values, we check what we inserted by executing a query on the database and using the `ResultsSet.next()` method to examine the results. We close the ResultSet and Statement objects in the reverse order in which they were instantiated.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.execute("select empno, empname from emp");

while (rs.next())  {
  System.out.println("Value = " + rs.getString("empno"));
  System.out.println("Value = " + rs.getString("empname"));
}
rs.close();
stmt.close();
```

Here we update the records we inserted, using another PreparedStatement.

```
String updsql = "update emp set empname = ? where empno = ?";
PreparedStatement pstmt1 = conn.prepareStatement(updsql);

for (int j = 0; j < 10; j++) {
  pstmt1.setInt(2, j);
  pstmt1.setString(1, "Person" + j);
  pstmt1.executeUpdate();
}
pstmt1.close();
```

In this example, we delete the records that we inserted. Again, we use a PreparedStatement and the `setInt()` method to select records for deletion.

```
String delsql = "delete from emp where empno = ?";
PreparedStatement pstmt2 = conn.prepareStatement(delsql);

for (int j = 0; j < 10; j++) {
  pstmt2.setInt(1, j);
  pstmt2.executeUpdate();
}
pstmt2.close();
```

Note that we close each Statement and PreparedStatement object when we are finished using it.

Creating and Using Stored Procedures and Functions

You can create, use, and drop stored procedures and functions with WebLogic JDBC. Use CallableStatement objects (subclass of PreparedStatement) with JDBC PreparedStatement "?" syntax to set parameters.

In this example, we drop several stored procedures and functions from a database, using a JDBC Statement that we close when we have finished:

```
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();
```

Use the JDBC Statement class to create a stored procedure or function, and use the JDBC CallableStatement class to execute them. Stored procedure input parameters are mapped to JDBC `IN` parameters, using the `CallableStatement.setXXX()` methods

like `setInt()`, and JDBC PreparedStatement "?" syntax. Stored procedure output parameters are mapped to JDBC `OUT` parameters, using the `CallableStatement.registerOutParameter()` methods and JDBC PreparedStatement "?" syntax. A parameter may be both `IN` and `OUT`, which requires both a `setXXX()` and a `registerOutParameter()` call to be done on the same parameter number. For the Sybase DBMS, the JDBC keyword `CALL` is used instead of the SQL Server keyword `EXECUTE`. For details, consult your Sybase documentation.

Here we create a Sybase stored procedure, using a JDBC Statement, to square an integer. Then we execute the stored procedure using a CallableStatement. We use the `registerOutParameter()` method to set an output parameter.

```
    Statement stmt1 = conn.createStatement();
    stmt1.execute("create procedure proc_squareInt " +
                  "(@field1 int, @field2 int output) as " +
  "begin select @field2 = @field1 * @field1 end");
    stmt1.close();

    CallableStatement cstmt1 =
      conn.prepareCall("{call proc_squareInt(?, ?)}");

    cstmt1.registerOutParameter(2, Types.INTEGER);
    for (int i = 0; i < 10; i++) {
      cstmt1.setInt(1, i);
      cstmt1.execute();
      System.out.println(i + " " + cstmt1.getInt(2));
    }
    cstmt1.close();
```

This example code shows how to create a Sybase stored function that returns the square of an integer. We execute the stored function with a CallableStatement, and use `registerOutParameter()` to register the return value.

```
    Statement stmt2 = conn.createStatement();
   stmt2.execute("create procedure func_squareInt (@field2 int) as
" +
                  "begin return @field1 * @field1 end");
    stmt2.close();

    CallableStatement cstmt2 =
      conn.prepareCall("{? = call func_squareInt(?)}");

    cstmt2.registerOutParameter(1, Types.INTEGER);
    for (int i = 0; i < 10; i++) {
      cstmt2.setInt(2, i);
      cstmt2.execute();
      System.out.println(i + " " + cstmt2.getInt(1));
```

```
    }
    cstmt2.close();
```

This example code shows how to create a Sybase stored procedure that returns the results of a SQL query. We execute the stored procedure with a CallableStatement and put the results into a ResultSet.

You must process all ResultSets returned by a stored procedure using the `Statement.execute()` and `Statement.getResultSet()` methods before `OUT` parameters and return status are available.

```
    Statement stmt3 = conn.createStatement();
    stmt3.executeUpdate("create procedure proc_getResults as " +
                        "begin select name from sysusers \n" +
        "select gid from sysusers end");
    stmt3.close();

    CallableStatement cstmt3 =
      conn.prepareCall("{call proc_getResults()}");

    boolean hasResultSet = cstmt3.execute();
    while (hasResultSet) {
      ResultSet rs = cstmt3.getResultSet();
      while (rs.next())
        System.out.println("Value: " + rs.getString(1));
      rs.close();
      hasResultSet = cstmt3.getMoreResults();
    }
    cstmt3.close();
```

Here is example code that shows how to create and call an Oracle stored procedure with Statement and CallableStatement objects. The process is similar to the Sybase 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 (int k = 0; k < 100; k++) {
```

```
      cstmt1.setInt(1, k);
      cstmt1.execute();
      System.out.println(k + " "
                         + cstmt1.getInt(1)
 + " " + cstmt1.getInt(2));
    }
    cstmt1.close();
```

Finally, here is sample code for creating and using an Oracle stored function with Statement and CallableStatement objects.

```
    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();

    // Use a stored function
    CallableStatement cstmt2 =
      conn.prepareCall("BEGIN ? := func_squareInt(?); END;");

    cstmt2.registerOutParameter(1, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt2.setInt(2, k);
      cstmt2.execute();
      System.out.println(k + " "
                         + cstmt2.getInt(1)
 + " " + cstmt2.getInt(2));
    }
    cstmt2.close();
```

Final Step. Closing the Connection and Disconnecting the T3Client

As with all other JDBC objects, you should close() the Connection, even if your login to the database fails; otherwise you risk exceeding the maximum number of logins. You should also disconnect() the WebLogic JDBCClient from the WebLogic Server.

Call the `close()` and `disconnect()` methods in a `try` block inside your `finally` block, and catch the appropriate Exceptions. Calling these methods within a `finally` block guarantees that they will be executed even if the code in your main `try` block throws an exception and does not complete. For example:

```
  finally {
    if (conn != null)
      try { conn.close();   } catch (SQLException sqe)  {}
```

```
    if (t3   != null)
      try { t3.disconnect();} catch (Exception e)       {}
  }
```

When a client disconnects with the `disconnect()` method, the WebLogic Server will do a dump stack with an EOF exception. You should ignore EOF exceptions that appear in your Netscape server log on `disconnect()`.

Code Summary

```
package examples.jdbc;

import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.common.*;

import java.util.Properties;

public class t3client1 {

  public static void main(String argv[]) {

    T3Client t3 = null;
    Connection conn = null;
    try {
      t3 = new T3Client("t3://bigbox:7001");
      t3.connect();

      Properties dbprops = new Properties();
      dbprops.put("user",                  "scott");
      dbprops.put("password",              "tiger");
      dbprops.put("server",               getParameter("server"));

      Properties t3props = new Properties();
      t3props.put("weblogic.t3",           t3);
      t3props.put("weblogic.t3.dbprops",   dbprops);

     // Note that the formats of the class name of the driver and the
     // URL are different; the class name uses dot-notation, and the
      // URL uses separating colons.
      t3props.put("weblogic.t3.driverClassName",
                  "weblogic.jdbc.oci.Driver");
      t3props.put("weblogic.t3.driverURL",
                  "jdbc:weblogic:oracle");
      t3props.put("weblogic.t3.cacheRows",
                  getParameter("cacheRows"));
```

```
      Class.forName("weblogic.jdbc.t3.Driver").newInstance();
      conn = DriverManager.getConnection("jdbc:weblogic:t3client",
                                         t3props);

      // Insert a series of records
      String inssql = "insert into emp (empno, empname) values (?,
?)";
      PreparedStatement pstmt = conn.prepareStatement(inssql);
      for (int j = 0; j < 10; j++)
        pstmt.setInt(1, i);
pstmt.setString(2, "John Smith");
pstmt.execute();
      }
      pstmt.close();

      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select empno, " +
                                       "empname from emp");

      while (rs.next())  {
        System.out.println("Value = " + rs.getString("empno"));
        System.out.println("Value = " + rs.getString("empname"));
      }

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

      // Update a series of records
      String updsql = "update emp set empname = ? where empno = ?";
      PreparedStatement pstmt1 = conn.prepareStatement(updsql);

      for (j = 0; j < 10; j++) {
        pstmt1.setInt(2, j);
        pstmt1.setString(1, "Person" + j);
        pstmt1.executeUpdate();
      }
      pstmt1.close();

      // Delete a series of records
      String delsql = "delete from emp where empno = ?";
      PreparedStatement pstmt2 = conn.prepareStatement(delsql);

      for (int j = 0; j < 10; j++) {
        pstmt2.setInt(1, j);
        pstmt2.executeUpdate();
      }
      pstmt2.close();

      // Create a stored procedures
```

```
    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();

    // Use a stored procedure
    CallableStatement cstmt1 =
      conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");

    cstmt1.registerOutParameter(2, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt1.setInt(1, k);
      cstmt1.execute();
      System.out.println(k + " "
                 + 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();

    // Use a stored function
    CallableStatement cstmt2 =
      conn.prepareCall("BEGIN ? := func_squareInt(?); END;");

    cstmt2.registerOutParameter(1, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt2.setInt(2, k);
      cstmt2.execute();
      System.out.println(k + " "
                 + cstmt2.getInt(1)
+ " " + cstmt2.getInt(2));
    }
    cstmt2.close();
  }
  finally {
    if (conn != null)
        try {conn.close();} catch (SQLException sqe) {}
```

```
        if (t3   != null)
            try {t3.disconnect();} catch (Exception e) {}
      }
    }
  }
```

Other WebLogic JDBC Features

WebLogic provides several features in its WebLogic jDrivers and in WebLogic JDBC to support database-specific strengths. All of the features supported in the two-tier drivers, and other features specific to multitier use, are available in the multitier environment in WebLogic JDBC.

Waiting on Oracle Resources

weblogic.jdbc.t3.Connection

With release 2.5, WebLogic supports Oracle's `oopt()` C functionality, 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. This functionality is described in section 4-97 of *The OCI Functions for C*.

The developer can set whether a client will wait for DBMS resources, or will receive an immediate exception. Here is an example, from the example `examples/jdbc/oracle/waiton.java`:

```
t3 = new T3Client("t3://bigbox:7001");
t3.connect();

java.util.Properties dbprops = new java.util.Properties();
dbprops.put("user",     "scott");
dbprops.put("password", "tiger");
dbprops.put("server",   "bigbox");

Properties t3props = new Properties();
t3props.put("weblogic.t3",                t3);
t3props.put("weblogic.t3.dbprops",        dbprops);
t3props.put("weblogic.t3.driverClassName",
            "weblogic.jdbc.oci.Driver");
t3props.put("weblogic.t3.driverURL",
```

```
            "jdbc:weblogic:oracle");
t3props.put("weblogic.t3.cacheRows",
            getParameter("cacheRows"));

Class.forName("weblogic.jdbc.t3.Driver").newInstance();

// You must cast the Connection as a weblogic.jdbc.oci.Connection
// to take advantage of this extension.
Connection conn =
  (weblogic.jdbc.oci.Connection)
     DriverManager.getConnection("jdbc:weblogic:t3", t3props);

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

Note that use of this 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.jdbc.oci.Connection`, and then call the `waitOnResources()` method.

Extended SQL

JavaSoft's JDBC specification includes a feature called SQL Extensions, or SQL Escape Syntax. All of the WebLogic jDriver JDBC drivers support Extended SQL. For more information, see the Developers Guide for your driver. For a listing, see WebLogic JDBC Options.

Oracle Array Fetches

WebLogic jDriver for Oracle offers support for Oracle array fetches by setting the two-tier JDBC Connection property, `weblogic.oci.cacheRows`. If you are using WebLogic jDriver for Oracle, this feature is also supported in WebLogic JDBC.

Take advantage of this feature in WebLogic JDBC by setting the two-tier property `weblogic.oci.cacheRows`.

Multibyte Character Set Support

WebLogic jDriver for Oracle also offers internationalization support (AL24UTFFSS/UTF-8), which is extended to WebLogic JDBC if you are using WebLogic jDriver for Oracle with the WebLogic Server. Full documentation on this feature is available in the Developers Guide for WebLogic jDriver for Oracle.

About WebLogic JDBC and Oracle NUMBER Columns

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

WebLogic jDriver for Oracle reliably converts the values in a column to the Java type requested when a WebLogic jDriver for Oracle 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 some problems. WebLogic jDriver for Oracle guarantees to return a Java object which will represent any value in a NUMBER column with no loss in precision. This means that a value of 1 can be returned in an `Integer`, but a value like 123434567890.123456789 can only be returned in a `BigDecimal`.

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

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

The multitier driver add another layer of complexity to this issue. By default, WebLogic fetches a configurable number of ResultSet rows of DBMS data before the multitier JDBC application asks for it, which improves performance as perceived by

the client. But because WebLogic does not know in advance what form the client will want the data, WebLogic prefetches rows generically, so that the data sent to a client application is of the same type in any one column of a ResultSet. When using prefetch, the `getObject()` method cannot be used for number types, because the WebLogic jDriver for Oracle in use by the WebLogic Server might return different Java types in a given column. Consequently, WebLogic prefetches all greater-than-integer numerical data in String form, to guarantee that no precision will be lost.

This has no effect on data retrieval in a WebLogic JDBC client application if the data is requested with `getInt()`, `getFloat()`, `getBigDecimal()`, etc. because WebLogic JDBC converts the String to the correct type. But for calls to the `getObject()` method, WebLogic JDBC returns the String it prefetched.

If you want `getObject()` to behave in the same way in your your multitier client application as in a two-tier WebLogic jDriver for Oracle application—that is, without prefetch, and potentially with mixed data types in the same column—turn off row caching in WebLogic by setting the Connection property weblogic.t3.cacheRows to zero (0).

Implementing with WebLogic JDBC and the JDBC-ODBC Bridge

Note:

This notated example shows how to use WebLogic JDBC to connect to any vendor's database using the JDBC-ODBC bridge as the two-tier driver.

Note: Using the JDBC-ODBC bridge to access a Microsoft Access database with Enterprise JavaBeans is not supported.

Step 1. Importing packages

- Step 2. Creating the T3Client
- Step 3. Connecting
  - Accessing data
  - Exception handling

- Final Step. Disconnecting and closing objects
- Code summary

Step 1. Importing packages

The same import packages are required for using the JDBC-ODBC bridge with WebLogic JDBC as with any other WebLogic JDBC class. They are:

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

Step 2. Creating the T3Client

The constructor for a WebLogic JDBC client takes one argument, the URL of the WebLogic Server, which includes the port on which it is listening for T3Client requests. The example places all of the initial work of the program in a try block.

```
try {
  t3 = new T3Client("t3://bigbox:7001");
  t3.connect();
```

You should call the `disconnect()` method on the t3Client in your finally block, even if the connection fails.

Step 3. Connecting

Use a java.util.Properties object to set parameters for connecting. We use one set of Properties for the WebLogic Server-to-DBMS connection (the two-tier connection), and another set of Properties for connection between the WebLogic JDBC client, the WebLogic Server, and the DBMS (the multitier connection). The two-tier Properties object itself is set as a multitier Property, and then the multitier Properties object is used as an argument for the Connection constructor. (details on setting Properties)

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

Properties t3props = new Properties();
t3props.put("weblogic.t3",                t3);
t3props.put("weblogic.t3.dbprops",        dbprops);
t3props.put("weblogic.t3.driverClassName",
```

```
                "sun.jdbc.odbc.JdbcOdbcDriver");
    t3props.put("weblogic.t3.driverURL",
                "jdbc:odbc:Oracle_on_SS2");
    t3props.put("weblogic.t3.cacheRows",      "10");

    Class.forName("weblogic.jdbc.t3.Driver").newInstance();
    conn = DriverManager.getConnection("jdbc:weblogic:t3client",
                                       t3props);

    checkForWarning(conn.getWarnings());
```

In the last line in this example, we check for vendor-specific warnings after the connection is established. Here is the code for the method `checkForWarning`. The method takes as an argument a SQLWarning object and displays information about the SQLState, the warning message, and the database vendor's error code. Note that there may be multiple warnings in a single SQLWarning object.

```
  private static boolean checkForWarning (SQLWarning warn)
       throws SQLException
  {
    boolean rc = false;
    if (warn != null) {
      System.out.println ("\n *** Warning ***\n");
      rc = true;
      while (warn != null) {
System.out.println ("SQLState: " +
    warn.getSQLState ());
System.out.println ("Message:  " +
    warn.getMessage ());
System.out.println ("Vendor:   " +
    warn.getErrorCode ());
System.out.println ("");
warn = warn.getNextWarning ();
      }
    }
    return rc;
  }
```

Accessing Data

Use the `getMetaData` method (in the Connection class) to retrieve database metadata. In this example, we display a few details about the database based on the metadata retrieved.

```
   DatabaseMetaData dma = conn.getMetaData();

   System.out.println("Connected to " + dma.getURL());
```

```
   System.out.println("Driver       " + dma.getDriverName());
   System.out.println("Version      " + dma.getDriverVersion());
   System.out.println("");
```

Use a `Statement` object to construct and execute a simple SQL select statement, and then retrieve the data into a ResultSet. Close the Statement and ResultSet objects when you have finished using them.

```
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
   dispResultSet (rs);

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

We display the results of the query with a private method that takes as its argument a ResultSet. The `dispResultSet()` method prints the column headings for the table using ResultSetMetaData and then displays the contents of the ResultSet in row major order.

```
  private static void dispResultSet (ResultSet rs)
       throws SQLException
  {
    int i;

    ResultSetMetaData rsmd = rs.getMetaData ();
    int numCols = rsmd.getColumnCount();

    for (i=1; i < = numCols; i++) {
      if (i > 1) System.out.print(",");
      System.out.print(rsmd.getColumnLabel(i));
    }
    System.out.println("");

    boolean more = rs.next ();
    while (more) {
      for (i=1; i<=numCols; i++) {
if (i > 1) System.out.print(",");
System.out.print(rs.getString(i));
      }
      System.out.println("");
      more = rs.next ();
    }
  }
```

Exception Handling

The SQLException is most interesting; we display the same information about a SQL error that we when we checked for SQLWarnings after instantiating a Connection. Note that there may be several SQL errors contained in a single SQLException.

```
catch (SQLException ex) {
  System.out.println ("\n*** SQLException caught ***\n");
  ex.printStackTrace();

  while (ex != null) {
     System.out.println ("SQLState: " +
   ex.getSQLState ());
     System.out.println ("Message:  " +
   ex.getMessage ());
     System.out.println ("Vendor:   " +
   ex.getErrorCode ());
     ex = ex.getNextException ();
     System.out.println ("");
  }
}
```

For all other exceptions, we merely print out a stack trace for debugging purposes.

```
catch (java.lang.Exception ex) {
  ex.printStackTrace ();
}
```

Final Step. Disconnecting and Closing Objects

Always close the Connection and disconnect the T3Client in a finally block to ensure proper cleanup.

```
finally {
  if (conn != null)
    try {conn.close();}    catch (Exception e) {;}
  if (t3   != null)
    try {t3.disconnect();} catch (Exception e) {;}
}
```

Code Summary

Here is a summary of the code discussed in this example.

```
package examples.jdbc.odbc;
```

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

class simpleselect {

  public static void main (String args[]) {

    String url   = "jdbc:odbc:Oracle_on_SS2";
    String query = "SELECT * FROM emp";

    T3Client   t3   = null;
    Connection conn = null;
    try {

      t3 = new T3Client("t3://bigbox:7001");
      t3.connect();

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

      Properties t3props = new Properties();
      t3props.put("weblogic.t3",               t3);
      t3props.put("weblogic.t3.dbprops",       dbprops);
      t3props.put("weblogic.t3.driverClassName",
                  "sun.jdbc.odbc.JdbcOdbcDriver");
      t3props.put("weblogic.t3.driverURL",     url);
      t3props.put("weblogic.t3.cacheRows",     "10");

      Class.forName("weblogic.jdbc.t3.Driver").newInstance();
      conn = DriverManager.getConnection("jdbc:weblogic:t3client",
                                         t3props);

      checkForWarning(conn.getWarnings());

      DatabaseMetaData dma = conn.getMetaData();

      System.out.println("Connected to " + dma.getURL());
      System.out.println("Driver       " + dma.getDriverName());
      System.out.println("Version     " + dma.getDriverVersion());
      System.out.println("");

      // Create a Statement object so we can submit
      // SQL statements to the driver
      Statement stmt = conn.createStatement();

      // Submit a query, creating a ResultSet object
      ResultSet rs = stmt.executeQuery(query);
```

```
      // Display all columns and rows from the result set
      dispResultSet (rs);

      // Close the result set and statement
      rs.close();
      stmt.close();

    }
    catch (SQLException ex) {

     // Catch SQLExceptions and display specific error information.
      System.out.println ("\n*** SQLException caught ***\n");
      ex.printStackTrace();

      while (ex != null) {
System.out.println ("SQLState: " +  ex.getSQLState ());
System.out.println ("Message:  " +  ex.getMessage ());
System.out.println ("Vendor:   " +  ex.getErrorCode ());
ex = ex.getNextException ();
System.out.println ("");
      }
    }
    catch (java.lang.Exception ex) {
      // For any other exception, just print out the StackTrace
      ex.printStackTrace();
    }
    finally {
      if (conn != null)
        try {conn.close();}    catch (SQLException sqe) {}
      if (t3   != null)
        try {t3.disconnect();} catch (Exception e) {}
    }
  }

  private static boolean checkForWarning (SQLWarning warn)
       throws SQLException
  {
    boolean rc = false;

    // Take an SQLWarning object and display its
    // warning messages.  Note that there could be
    // multiple warnings chained together.

    if (warn != null) {
      System.out.println ("\n *** Warning ***\n");
      rc = true;
      while (warn != null) {
System.out.println ("SQLState: " + warn.getSQLState ());
```

```
System.out.println ("Message:  " + warn.getMessage ());
System.out.println ("Vendor:   " + warn.getErrorCode ());
System.out.println ("");
warn = warn.getNextWarning ();
      }
    }
    return rc;
  }

  private static void dispResultSet (ResultSet rs)
       throws SQLException
  {
    int i;
    ResultSetMetaData rsmd = rs.getMetaData();
    int numCols = rsmd.getColumnCount();

    for (i=1; i<=numCols; i++) {
      if (i > 1) System.out.print(",");
      System.out.print(rsmd.getColumnLabel(i));
    }
    System.out.println("");

    boolean more = rs.next();
    while (more) {
      for (i=1; i<=numCols; i++) {
if (i > 1) System.out.print(",");
System.out.print(rs.getString(i));
      }
      System.out.println("");
      more = rs.next();
    }
  }
}
```

Using URLs to Set Properties For a JDBC Connection Using the T3 Driver

Where URLs are Used

URLs -- uniform resource locators -- are tools for identifying and locating resources over a network. The JDBC specification makes general recommendations for the use of URLs within JDBC. We have implemented these recommendations in a way that is used consistently throughout all of our products, and that abides by the current URL guidelines (at www.w3.org).

With WebLogic products, you can use a URL to specify parameters needed to make a WebLogic JDBC Connection.

How WebLogic URLs are Structured

Specifying a Connection with a Properties Object and a URL

The WebLogic JDBC drivers use a `java.utils.Properties` object to set properties that are used to open a JDBC Connection between your client and the target database. The WebLogic JDBC drivers also use a URL, as described in the JDBC specification, to identify the JDBC driver. The Properties object and the URL are used as arguments for the `DriverManager.getConnection()` method or the `Driver.connect()` method.

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

The usual process follows this model:

Construct a `java.util.Properties` object for specifying certain information like username and password.

Call the `Class.forName().newInstance()` method with the classname of the JDBC driver to load it and cast it to a `java.sql.Driver` object.

Create a connection with the `Driver.connect()` method, which takes two arguments, the URL of the JDBC driver and a set of properties.

Here is a working code snippet that illustrates creating the Properties object, calling the `Class.forName().newInstance()` method, and creating a Connection object:

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

The Properties object that contains the username, password, and servername for the two-tier connection becomes part of another Properties object that we use to set parameters for the multitier WebLogic JDBC connection.

```
  Properties t3props = new Properties();
  t3props.put("weblogic.t3",                t3);
  t3props.put("weblogic.t3.dbprops",        dbprops);
  t3props.put("weblogic.t3.driverClassName",
"weblogic.jdbc.oci.Driver");
  t3props.put("weblogic.t3.driverURL",
"jdbc:weblogic:oracle");
```

All of the information for both the two-tier and multitier connections is then passed to the Driver when the JDBC Connection is instantiated by calling the `Driver.connect()` method. This method takes two arguments, a String URL and a `java.util.Properties` object.

```
 Driver myDriver = (Driver)
  Class.forName("weblogic.jdbc.t3.Driver").newInstance();
  Connection conn = Driver.connect("jdbc:weblogic:t3", t3props);
```

Specifying a WebLogic JDBC Connection with a Single URL

Some development environments (like Sybase's PowerJ) use the Properties object exclusively for username and database. That restriction means that you must use the URL passed as the first argument to the `DriverManager.getConnection()` method to set all of the other parameters required for a WebLogic JDBC connection, including the servername, as well as all of the `weblogic.t3` properties.

For such environments, we allow you to supply all of the other information needed for a WebLogic JDBC Connection by constructing a long URL that is patterned after an RFC 1630-style query string.

Here is the syntax for WebLogic URLs. The property name is shown in bold, and a sample value follows it.

`jdbc:weblogic:t3 (with JDK 1.1)`

First, set the URL for the WebLogic JDBC driver. (In technical terms, the URL of the WebLogic JDBC driver is the *scheme of the URL.)*

Please note that WebLogic no longer supports JDK 1.0.2.

The WebLogic JDBC driver URL should be followed by a question mark (?), after which follows a series of name-value pairs. Each name-value pair is separated by an ampersand (&). (The example shows these delimiting characters in a different color.) Arguments that come after the "?" are shown here in an arbitrary order, and can occur in the URL in any order.

`weblogic.t3.serverURL=t3://localhost:7001`

Sets the URL of the WebLogic Server.

`weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO`

Sets the URL of the two-tier JDBC driver. The syntax should include the information that you would set as the "server" property. For example, `jdbc:weblogic:oracle:DEMO` indicates a WebLogic JDBC driver for an Oracle database on the host "DEMO".

`weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver`

Sets the class name of the two-tier JDBC driver. Note that class names are always indicated as the full package name of the class, in dot-notation format.

`weblogic.t3.cacheRows=10`

Sets the rows to be cached on the WebLogic Server.

`weblogic.t3.connectionPoolID=eng`

Sets the connection pool ID. This ID must be registered in the `weblogic.properties` file. If you use a JDBC connection from a connection pool, other information in the URL is unnecessary. For more information, read up on connection pools in Using WebLogic JDBC.

There are other optional properties that you may add to this list. For details on properties, see Setting properties for connecting in the Developers Guide.

Here is an example of specifying a URL for a connection to an Oracle database named DEMO, with a WebLogic Server running on port 7001 of a host "toyboat.toybox.com," with the cacheRows property set to 25. This example assumes that you have set "username" and "password" properties for access to the database with a Properties object. The characters "?" and "&" have special meanings in a URL and are set off here in red. For simplicity, the different parts of the URL are displayed in different lines; in reality, this URL is one long string.

```
jdbc:weblogic:t3?
weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver&
weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO&
weblogic.t3.cacheRows=25
```

Shortcuts

Given certain pieces of information, we can infer other details that allow you to simplify the URL.

For example, you can supply more information first piece of the URL, the WebLogic JDBC driver URL, like database vendor and DBMS host, that negates the need for the `weblogic.t3.driverURL`, as shown here:

```
jdbc:weblogic:t3:oracle:DEMO
```

rather than:

```
jdbc:weblogic:t3?weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO
```

If you are using one of the drivers in WebLogic jDriver group, you can infer the class name of the driver from the first piece of the URL that we shortened in the example above, or from the property `weblogic.t3.driverURL`. We map the vendor name in either URL to a driver in the WebLogic jDriver group, as shown in this table:

URL	Inferred driver
`jdbc:weblogic:oracle`	`weblogic.jdbc.oci.Driver`
`jdbc:weblogic:mssqlserver4`	`weblogic.jdbc.mssqlserver4.Driver`
`jdbc:weblogic:informix4`	`weblogic.jdbc.informix4.Driver`

These shortcuts together reduce the URL used in the example to the following:

```
jdbc:weblogic:t3:oracle:DEMO?
 weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
 weblogic.t3.cacheRows=25
```

Quoting Metacharacters in a URL

URL syntax allows only a subset of the graphic printable characters of the US-ASCII coded character set, specifically the letters A-Z (both upper and lower case), the digits, and the characters $-_.+!*'()" may be used in a URL. Any other characters should be represented by a character triplet that is the character "%" followed by two hexadecimal digits ("0123456789ABCDEF") which form the hexadecimal representation of the character.

Some characters may be reserved by a URL scheme; those characters include ; / ? : @ = and &. These must always be encoded when used outside their reserved purpose in a URL.

The following characters are reserved in the WebLogic URL scheme. You must use a character triplet to represent these characters in a URL for anything other than a reserved purpose:

%26 (&)

%3D (=)

%3F (?)

%2F (/)

%3A (:)

Using IDEs and Wizards

You can also use URLs with IDEs -- integrated development environments or wizards -- like Sybase's PowerJ. If an IDE requires the fully qualified classname and a URL, here is how the classname for WebLogic's JDBC driver should be entered:

```
JDBC Driver:
 weblogic.jdbc.t3.Driver
```

The characters "?" and "&" have special meanings in a URL and are set off here in red. For simplicity, the different parts of the URL are displayed in different lines; in reality, this URL is one long string. Following is the URL for the PowerJ database wizard.

```
Data Source URL:
 jdbc:weblogic:t3?
 weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
 weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver&
 weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO&
 weblogic.t3.cacheRows=25
```