

Oracle® Fusion Middleware

Developing JDBC Applications for Oracle WebLogic Server



12c (12.2.1.4.0)
E90815-02
October 2020

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Copyright © 2007, 2020, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

	Preface	
	Documentation Accessibility	ix
	Conventions	ix
1	Introduction and Roadmap	
	Document Scope and Audience	1-1
	Guide to this Document	1-1
	Related Documentation	1-2
	JDBC Samples and Tutorials	1-2
	Avitek Medical Records Application (MedRec)	1-2
	JDBC Examples in the WebLogic Server Distribution	1-2
	New and Changed Features in This Release	1-2
2	Using WebLogic JDBC in an Application	
	Getting a Database Connection from a DataSource Object	2-1
	Importing Packages to Access DataSource Objects	2-1
	Obtaining a Client Connection Using a DataSource	2-1
	Possible Exceptions When a Connection Request Fails	2-3
	Pooled Connection Limitation	2-3
	Getting a Connection from an Application-Scoped Data Source	2-4
3	Using DataSource Resource Definitions	
	Using Java EE DataSource Resource Definitions	3-1
	Creating DataSource Resource Definitions Using Annotations	3-2
	Creating DataSource Resource Definitions Using Deployment Descriptors	3-3
	Using WebLogic Configuration Attributes	3-3
	Implementation Considerations When Using DataSource Resource Definitions	3-6
	Naming Conventions	3-6
	WebLogic Data Source Naming Conventions	3-6
	Java EE Data Source Naming Conventions	3-7

Mapping the Java EE DataSource Resource Definition to WebLogic Data Source Resources	3-7
Configuring Active GridLink DataSource Resource Definitions	3-8
Using an Encrypted Password in a DataSourceDefinition	3-10
Additional Considerations	3-11
Using Data Sources in Clients	3-11
Additional Resources	3-12

4 Performance Tuning Your JDBC Application

WebLogic Performance-Enhancing Features	4-1
How Pooled Connections Enhance Performance	4-1
Caching Statements and Data	4-1
Designing Your Application for Best Performance	4-1
Process as Much Data as Possible Inside the Database	4-2
Use Built-in DBMS Set-based Processing	4-2
Make Your Queries Smart	4-2
Make Transactions Single-batch	4-4
Never Have a DBMS Transaction Span User Input	4-4
Use In-place Updates	4-5
Keep Operational Data Sets Small	4-5
Use Pipelining and Parallelism	4-5

5 Using WebLogic-branded DataDirect Drivers

Using DataDirect Documentation	5-1
JDBC Specification Compliance	5-1
Installation	5-2
Supported Drivers and Databases	5-2
Connecting Through WebLogic JDBC Data Sources	5-3
Developing Your Own JDBC Code	5-3
Specifying Connection Properties	5-3
Using IP Addresses	5-3
Required Permissions for the Java Security Manager	5-3
For MS SQLServer Users	5-4
Installing MS SQLServer XA DLLs	5-4
Using instjdbc.sql with MS SQLServer	5-4

6 Using WebLogic Wrapper Drivers

Using the WebLogic RMI Driver (Deprecated)	6-1
RMI Driver Client Interoperability	6-1

Security Considerations for WebLogic RMI Drivers	6-2
Setting Up WebLogic Server to Use the WebLogic RMI Driver	6-2
Sample Client Code for Using the RMI Driver	6-2
Import the Required Packages	6-2
Get the Database Connection	6-3
Using a JNDI Lookup to Obtain the Connection	6-3
Using Only the WebLogic RMI Driver to Obtain a Database Connection	6-4
Row Caching with the WebLogic RMI Driver	6-5
Important Limitations for Row Caching with the WebLogic RMI Driver	6-5
Limitations When Using Global Transactions	6-6
Using the WebLogic JTS Driver (Deprecated)	6-7
Sample Client Code for Using the JTS Driver	6-7

7 Using API Extensions in JDBC Drivers

Using API Extensions to JDBC Interfaces	7-1
Sample Code for Accessing API Extensions to JDBC Interfaces	7-1
Import Packages to Access API Extensions	7-1
Get a Connection	7-2
Cast the Connection as a Vendor Connection	7-2
Use API Extensions	7-2
Using API Extensions for Oracle JDBC Types	7-2
Sample Code for Accessing Oracle Thin Driver Extensions to JDBC Interfaces	7-4
Programming with Arrays	7-5
Import Packages to Access Oracle Extensions	7-5
Establish the Connection	7-5
Creating an Array in the Database	7-6
Getting an Array	7-6
Updating an Array in the Database	7-7
Using Oracle Array Extension Methods	7-7
Programming with Structs	7-7
Creating Objects in the Database	7-8
Getting Struct Attributes	7-8
Using OracleStruct Extension Methods	7-9
Using a Struct to Update Objects in the Database	7-9
Programming with Refs	7-10
Creating a Ref in the Database	7-10
Getting a Ref	7-10
Using WebLogic OracleRef Extension Methods	7-11
Updating Ref Values	7-11
Programming with Large Objects	7-12

Creating Blobs in the Database	7-12
Updating Blobs in the Database	7-12
Using OracleBlob Extension Methods	7-12
Programming with Clob Values	7-13
Transaction Boundaries Using LOBs	7-13
Recovering LOB Space	7-13
Programming with Opaque Objects	7-13
Using Batching with the Oracle Thin Driver	7-14
Using the Java Security Manager with the Oracle Thin Driver	7-15

8 Getting a Physical Connection from a Data Source

Opening a Connection	8-1
Closing a Connection	8-2
Remove Infected Connections Enabled is True	8-3
Remove Infected Connections Enabled is False	8-4
Limitations for Using a Physical Connection	8-4

9 Using RowSets with WebLogic Server

Deprecation of weblogic.jdbc.rowsets	9-1
About RowSets	9-1
Types of RowSets	9-2
Programming with RowSets	9-2
CachedRowSets	9-3
Characteristics	9-4
Special Programming Considerations and Limitations for CachedRowSets	9-4
Entire RowSet Query Results Stored in Memory	9-4
Data Contention	9-4
Code Example	9-4
Importing Classes and Interfaces for a CachedRowSet	9-6
Creating a CachedRowSet	9-6
Setting CachedRowSet Properties	9-6
Database Connection Options	9-6
Populating a CachedRowSet	9-7
Setting CachedRowSet MetaData	9-8
Working with Data in a CachedRowSet	9-8
Getting Data from a Row in a RowSet	9-8
Updating a Row in a RowSet	9-8
Inserting a Row in a RowSet	9-9
Deleting a Row in a RowSet	9-9

Synchronizing RowSet Changes with the Database	9-10
RowSet MetaData Settings for Database Updates	9-10
WebLogic RowSet Extensions for Working with MetaData	9-11
executeAndGuessTableName and executeAndGuessTableNameAndPrimaryKeys	9-11
Setting Table and Primary Key Information Using the MetaData Interface	9-12
Setting the Write Table	9-12
RowSets and Transactions	9-12
Integrating with JTA Global Transactions	9-13
Behavior of Rowsets Using Global Transactions	9-13
Using Local Transactions	9-13
Behavior of Rowsets Using Local Transactions	9-13
Reusing a WebLogic RowSet After Completing a Transaction	9-14
FilteredRowSets	9-14
FilteredRowSet Characteristics	9-15
Special Programming Considerations	9-15
RowSet Filters are Not Cumulative	9-15
No Pending Changes Before Setting or Changing a Filter	9-15
FilteredRowSet Code Example	9-16
Importing Classes and Interfaces for FilteredRowSets	9-17
Creating a FilteredRowSet	9-17
Setting FilteredRowSet Properties	9-18
Database Connection Options for a FilteredRowSet	9-18
Populating a FilteredRowSet	9-18
Setting FilteredRowSet MetaData	9-18
Setting the Filter for a FilteredRowSet	9-18
User-Defined RowSet Filter	9-18
WebLogic SQL-Style Filter	9-19
Working with Data in a FilteredRowSet	9-20
WebRowSets	9-20
Special Programming Considerations	9-20
JoinRowSets	9-21
JDBCRowSets	9-22
Handling SyncProviderExceptions with a SyncResolver	9-22
RowSet Data Synchronization Conflict Types	9-23
SyncResolver Code Example	9-24
Getting a SyncResolver Object	9-25
Navigating in a SyncResolver Object	9-25
Setting the Resolved Value for a RowSet Data Synchronization Conflict	9-26
Synchronizing Changes	9-26
WLCachedRowSets	9-26

SharedRowSets	9-27
SortedRowSets	9-27
SQLPredicate, a SQL-Style RowSet Filter	9-28
What is SQLPredicate?	9-28
SQLPredicate Grammar	9-28
Code Example	9-28
Optimistic Concurrency Policies	9-29
VERIFY_READ_COLUMNS	9-30
VERIFY_MODIFIED_COLUMNS	9-30
VERIFY_SELECTED_COLUMNS	9-30
VERIFY_NONE	9-31
VERIFY_AUTO_VERSION_COLUMNS	9-31
VERIFY_VERSION_COLUMNS	9-31
Optimistic Concurrency Control Limitations	9-32
Choosing an Optimistic Policy	9-32
Performance Options	9-32
JDBC Batching	9-32
Batching Limitations with and Oracle Database	9-33
Group Deletes	9-33

10 Troubleshooting JDBC

Problems with Oracle Database on UNIX	10-1
Thread-related Problems on UNIX	10-1
Closing JDBC Objects	10-1
Abandoning JDBC Objects	10-2
Using Microsoft SQL Server with Nested Triggers	10-2
Exceeding the Nesting Level	10-2
Using Triggers and EJBs	10-3

Preface

This preface describes the document accessibility features and conventions used in this guide—*Developing JDBC Applications for Oracle WebLogic Server*.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Accessible Access to Oracle Support

Oracle customers who have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction and Roadmap

This chapter describes the contents and organization of this guide—*Developing JDBC Applications for Oracle WebLogic Server*.

This chapter includes the following sections:

- [Document Scope and Audience](#)
- [Guide to this Document](#)
- [Related Documentation](#)
- [JDBC Samples and Tutorials](#)
- [New and Changed Features in This Release](#)

Document Scope and Audience

This document is a resource for software developers and system administrators who develop and support applications that use the Java Database Connectivity (JDBC) API. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server. The topics in this document are relevant during the evaluation, design, development, pre-production, and production phases of a software project.

It is assumed that the reader is familiar with Java EE and JDBC concepts. This document emphasizes the value-added features provided by WebLogic Server JDBC and key information about how to use WebLogic Server features and facilities to get an JDBC application up and running.

Guide to this Document

- This chapter, [Introduction and Roadmap](#), introduces the organization of this guide.
- [Using WebLogic JDBC in an Application](#), explains how to use a JDBC connection in your application.
- [Using DataSource Resource Definitions](#), explains how to use a DataSource resource definition in your application.
- [Performance Tuning Your JDBC Application](#), describes how to design JDBC connection usage in your applications for the best performance.
- [Using WebLogic-branded DataDirect Drivers](#), describes how to use WebLogic-branded DataDirect drivers in your applications.
- [Using WebLogic Wrapper Drivers](#), describes how to use some alternative drivers for getting a JDBC connection from a data source.
- [Using API Extensions in JDBC Drivers](#), describes special programming considerations for third-party drivers in your applications.
- [Using RowSets with WebLogic Server](#), describes how to use rowsets in your applications.

- [Troubleshooting JDBC](#), describes some common JDBC problems and solutions.

Related Documentation

This document contains JDBC-specific programming information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Administering JDBC Data Sources for Oracle WebLogic Server* is a guide to JDBC configuration and management for WebLogic Server.
- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications in development and production environments.

JDBC Samples and Tutorials

In addition to this document, Oracle provides a variety of JDBC code samples that show JDBC configuration and API use, and provide practical instructions on how to perform key JDBC development tasks.

Avitek Medical Records Application (MedRec)

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights Oracle-recommended best practices. MedRec is optionally installed with the WebLogic Server installation. You can start MedRec from the `ORACLE_HOME\user_projects\domains\medrec` directory, where `ORACLE_HOME` is the directory you specified as the Oracle Home when you installed Oracle WebLogic Server. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

JDBC Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in the `ORACLE_HOME\wlserver\samples\server` directory, where `ORACLE_HOME` represents the directory where you installed WebLogic Server. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

2

Using WebLogic JDBC in an Application

Learn how to use the WebLogic Server Administration Console to enable, configure, and monitor features of WebLogic Server, including JDBC generic data sources, multi data sources, or Active GridLink data sources. You can do the same tasks programmatically using the JMX API and the WebLogic Scripting Tool (WLST). After configuring JDBC connectivity components, you can use them in your applications.

- [Getting a Database Connection from a DataSource Object](#)
- [Pooled Connection Limitation](#)
- [Getting a Connection from an Application-Scoped Data Source](#)

See [Configuring JDBC Data Sources in *Administering JDBC Data Sources for Oracle WebLogic Server*](#).

Getting a Database Connection from a DataSource Object

Learn how to request a database connection from a DataSource object.

- [Importing Packages to Access DataSource Objects](#)
- [Obtaining a Client Connection Using a DataSource](#)
- [Possible Exceptions When a Connection Request Fails](#)

Importing Packages to Access DataSource Objects

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

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

Obtaining a Client Connection Using a DataSource

To obtain a connection for a JDBC client, use a Java Naming and Directory Interface (JNDI) lookup to locate the DataSource object, as shown in this code fragment.

 **Note:**

When using a JDBC connection in a client-side application, the *exact same* JDBC driver classes must be in the CLASSPATH on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

```

Context ctx = null;
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,
        "t3://hostname:port");
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
        = (javax.sql.DataSource) ctx.lookup ("myDataSource");
    conn = ds.getConnection();
    // You can now use the conn object to create
    // Statements and retrieve result sets:
    stmt = conn.createStatement();
    stmt.execute("select * from someTable");
    rs = stmt.getResultSet();
    ...
    //Close JDBC objects as soon as possible
    stmt.close();
    stmt=null;
    conn.close();
    conn=null;
}
catch (Exception e) {
    // a failure occurred
    log message;
}
finally {
    try {
        ctx.close();
    } catch (Exception e) {
        log message; }
    try {
        if (rs != null) rs.close();
    } catch (Exception e) {
        log message; }
    try {
        if (stmt != null) stmt.close();
    } catch (Exception e) {
        log message; }
    try {
        if (conn != null) conn.close();
    } catch (Exception e) {
        log message; }
}
}

```

(Substitute the correct hostname and port number for your WebLogic Server.)



Note:

The code above uses one of several available procedures for obtaining a JNDI context. For more information on JNDI, see *WebLogic Server JNDI in Developing JNDI Applications for Oracle WebLogic Server*.

Possible Exceptions When a Connection Request Fails

The `weblogic.jdbc.extensions` package includes the following exceptions that can be thrown when an application request fails. Each exception extends `java.sql.SQLException`.

- `ConnectionDeadSQLException`—generated when an application request to get a connection fails because the connection test on the reserved connection failed. This typically happens when the database server is unavailable.
- `ConnectionUnavailableSQLException`—generated when an application request to get a connection fails because there are currently no connections available in the pool to be allocated. This is a transient failure, and is generated if all connections in the pool are currently in use. It can also be thrown when connections are unavailable because they are being tested.
- `PoolDisabledSQLException`—generated when an application request to get a connection fails because the JDBC Data Source has been administratively disabled.
- `PoolLimitSQLException`—generated when an application request to get a connection fails due to a configured threshold of the data source, such as `HighestNumWaiters`, `ConnectionReserveTimeoutSeconds`, and so forth.
- `PoolPermissionsSQLException`—generated when an application request to get a connection fails a (security) authentication or authorization check.

Pooled Connection Limitation

When using pooled connections in a data source, it is possible to execute DBMS-specific SQL code that will alter the database connection properties in a way which WebLogic Server and the JDBC driver will be unaware of. When the connection is returned to the pool, the characteristics of the connection may not be set back to a valid state.

 **Note:**

For example, with a Sybase DBMS, if you use a statement such as `"set rowcount 3 select * from y"`, the connection will only ever return a maximum of 3 rows from any subsequent query on this connection. When the connection is returned to the pool and then reused, the next user of the connection will still only get 3 rows returned, even if the table being selected from has 500 rows.

When using pooled connections in a data source, it is possible to execute DBMS-specific SQL code that will alter the database connection properties and that WebLogic Server and the JDBC driver will be unaware of. When the connection is returned to the pool, the characteristics of the connection may not be set back to a valid state. For example, with a Sybase DBMS, if you use a statement such as `"set rowcount 3 select * from y"`, the connection will only ever return a maximum of 3 rows from any subsequent query on this connection. When the connection is returned to the pool and

then reused, the next user of the connection will still only get 3 rows returned, even if the table being selected from has 500 rows.

In most cases, there is standard JDBC code that can accomplish the same result. In this example, you could use `setMaxRows()` instead of `set rowcount`. Oracle recommends that you use the standard JDBC code instead of the DBMS-specific SQL code. When you use standard JDBC calls to alter the connection, WebLogic Server returns the connection to a standard state when the connection is returned to the data source.

If you use vendor-specific SQL code that alters the connection, you must set the connection back to an acceptable state before returning the connection to the pool.

Getting a Connection from an Application-Scoped Data Source

To get a connection from JDBC module packaged with an enterprise application, you look up the data source defined in the JDBC module in the local environment or in the JNDI tree and then request a connection from the data source or multi data source.

To get a connection from an application-scoped data source, see *Getting a Database Connection from a Packaged JDBC Module* in *Administering JDBC Data Sources for Oracle WebLogic Server*.

3

Using DataSource Resource Definitions

Data source resources are used to define a set of properties required to identify and access a database through the JDBC API. Learn how to create and use Java EE DataSource resource definitions.

- [Using Java EE DataSource Resource Definitions](#)
- [Using WebLogic Configuration Attributes](#)
- [Implementation Considerations When Using DataSource Resource Definitions](#)
- [Using Data Sources in Clients](#)
- [Additional Resources](#)

Using Java EE DataSource Resource Definitions

Data source resources are used to define a set of properties required to identify and access a database through the JDBC API. These properties include information such as the URL of the database server, the name of the database, and the network protocol to use to communicate with the server. You can declare data source definitions by creating data source resource definitions using annotations or deployment descriptor.

DataSource objects are registered with the Java Naming and Directory Interface (JNDI) naming service so that applications can use the JNDI API to access a DataSource object to make a connection with a database.

Prior to Java EE 7, DataSource resources were created administratively as described in *Configuring WebLogic JDBC Resources in Administering JDBC Data Sources for Oracle WebLogic Server*. Java EE 7 provides the option to programmatically define DataSource resources for a more flexible and portable method of database connectivity.

The `name` element uniquely identifies a DataSource and is registered with JNDI. The value specified in the `name` element begins with a namespace scope. Java EE 7 includes the following scopes:

- `java:comp`—Names in this namespace have per-component visibility.
- `java:module`—Names in this namespace are shared by all components in a module, for example, the EJB components defined in an `ejb-jar.xml` file.
- `java:app`—Names in this namespace are shared by all components and modules in an application, for example, the application-client, web, and EJB components in an `.ear` file.
- `java:global`—Names in this namespace are shared by all the applications in the server.

You can programmatically declare data source definitions using one of the following methods:

- [Creating DataSource Resource Definitions Using Annotations](#)

- [Creating DataSource Resource Definitions Using Deployment Descriptors](#)

Creating DataSource Resource Definitions Using Annotations

The `javax.annotation.sql` package provides `@DataSourceDefinition` and `@DataSourceDefinitions` for defining DataSource resource definitions in application component classes such as application clients, servlets, or Enterprise JavaBeans (EJB).

When the DataSource resource is injected, a DataSource object is created and registered with JNDI. Use annotation elements to configure the DataSource object. You can specify additional Java EE and WebLogic configuration attributes in the `properties` element of the annotation. See [Using WebLogic Configuration Attributes](#).

Use `@DataSourceDefinition` to create a single datasource definition. For example:

```
. . .

@DataSourceDefinition(
    name = "java:module/ExampleDS",
    className = "org.apache.derby.jdbc.ClientDataSource",
    portNumber = 1527,
    serverName = "localhost",
    databaseName = "exampleDB",
    user = "examples",
    password = "examples",
    properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM
SYS.SYSTABLES"})

@WebServlet("/dataSourceServlet")
public class DataSourceServlet extends HttpServlet {

    . . .

    @Resource(lookup = "java:module/ExampleDS")

    . . .
```

Use the `@DataSourceDefinitions` to create multiple datasource definitions. For example:

```
. . .

@DataSourceDefinitions(
    value = {

        @DataSourceDefinition(name = "java:app/env/DS1",
            minPoolSize = 0,
            initialPoolSize = 0,
            className = "org.apache.derby.jdbc.ClientXADataSource",
            portNumber = 1527,
            serverName = "localhost",
            user = "examples",
            password = "examples",
            databaseName = "exampleDB",
            properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM
SYS.SYSTABLES"})
    },

    @DataSourceDefinition(name = "java:comp/env/DS2",
```

```

        minPoolSize = 0,
        initialPoolSize = 0,
        className = "org.apache.derby.jdbc.ClientDataSource",
        portNumber = 1527,
        serverName = "localhost",
        user = "examples",
        password = "examples",
        databaseName = "examplesDB",
        properties={"create=true", "weblogic.TestTableName=SQL SELECT 1 FROM
SYS.SYSTABLES"}
    )
}
)
. . .

```

For a complete example, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Creating DataSource Resource Definitions Using Deployment Descriptors

You can create `DataSource` resource definitions using deployment descriptors in `application.xml`, `application-client.xml`, `web.xml`, and `ejb-jar.xml` files. For example:

```

. . .
<data-source>
  <name>java:module/ExampleDS</name>
  <class-name>org.apache.derby.jdbc.ClientDataSource</class-name>
  <server-name>localhost</server-name>
  <port-number>1527</port-number>
  <database-name>exampleDB</database-name>
  <user>examples</user>
  <password>examples</password>
  <property>
    <name>create</name>
    <value>true</value>
  </property>
  <property>
    <name>weblogic.TestTableName</name>
    <value>SQL SELECT 1 FROM SYS.SYSTABLES</value>
  </property>
</data-source>
. . .

```

Using WebLogic Configuration Attributes

The Java EE 7 Definition annotation `@DataSourceDefinition` provides a basic standard set of configuration attributes. Oracle extends support for WebLogic Server's rich set of configuration attributes by supporting proprietary attributes using the `property` element.

 **Note:**

Consider the following limitations when using WebLogic Server proprietary attributes in the `property` element. WebLogic Server proprietary attributes:

- Cannot be used to configure a Multi data source. It is not possible to embed a Multi data source in a EAR or WAR file.
- Do not overlap `@DataSourceDefinition` annotation elements.
- Do not include the data source level attributes `name` and `version`.

[Table 3-1](#) summarizes WebLogic Server's extended support for Data Source configuration attributes by mapping `Weblogic.Attribute` Name property values to WebLogic configuration elements. For an example of a `DataSource` resource definition using WebLogic configuration elements, see [Configuring Active GridLink DataSource Resource Definitions](#).

Table 3-1 WebLogic Configuration Attributes

Weblogic.Attribute Name	WebLogic Element
AffinityPolicy	<code>JDBCOracleParams.setAffinityPolicy()</code>
AlgorithmType	<code>JDBCDataSourceParams.setAlgorithmType()</code>
CapacityIncrement	<code>JDBCConnectionPoolParams.setCapacityIncrement()</code>
ConnectionCreationRetr yFrequencySeconds	<code>JDBCConnectionPoolParams.setConnectionCreationRetr yFrequencySeconds()</code>
ConnectionPoolFailover CallbackHandler	<code>JDBCDataSourceParams.setConnectionPoolFailoverCall backHandler()</code>
ConnectionReserveTimeo utSeconds	<code>JDBCConnectionPoolParams.setConnectionReserveTimeo utSeconds()</code>
CredentialMappingEnabl e	<code>JDBCConnectionPoolParams.setCredentialMappingEnabl ed()</code>
DataSourceList	<code>JDBCDataSourceParams.setDataSourceList()</code>
DriverInterceptor	<code>JDBCConnectionPoolParams.setDriverInterceptor()</code>
FailoverRequestIfBusy	<code>JDBCDataSourceParams.setFailoverRequestIfBusy()</code>
FanEnabled	<code>JDBCOracleParams.setFanEnabled()</code>
GlobalTransactionsProt ocol	<code>JDBCDataSourceParams.setGlobalTransactionsProtoco l()</code>
HighestNumWaiters	<code>JDBCConnectionPoolParams.setHighestNumWaiters()</code>
IdentityBasedConnectio nPoolingEnabled	<code>JDBCConnectionPoolParams.setIdentityBasedConnectio nPoolingEnabled()</code>
IgnoreInUseConnections Enabled	<code>JDBCConnectionPoolParams.setIgnoreInUseConnections Enabled()</code>
InactiveConnectionTime outSeconds	<code>JDBCConnectionPoolParams.setInactiveConnectionTime outSeconds()</code>
InitSql	<code>JDBCConnectionPoolParams.setInitSql()</code>
JDBCXADebugLevel	<code>JDBCConnectionPoolParams.setJDBCXADebugLevel()</code>

Table 3-1 (Cont.) WebLogic Configuration Attributes

Weblogic.Attribute Name	WebLogic Element
KeepConnAfterLocalTx	JDBCDataSourceParams.setKeepConnAfterLocalTx()
KeepLogicalConnOpenOnRelease	JDBCXAParams.setKeepLogicalConnOpenOnRelease()
KeepXaConnTillTxComplete	JDBCXAParams.setKeepXaConnTillTxComplete()
LoginDelaySeconds	JDBCConnectionPoolParams.setLoginDelaySeconds()
NeedTxCtxOnClose	JDBCXAParams.setNeedTxCtxOnClose()
NewXaConnForCommit	JDBCXAParams.setNewXaConnForCommit()
OnsNodeList	JDBCOracleParams.setOnsNodeList()
OnsWalletFile	JDBCOracleParams.setOnsWalletFile()
OnsWalletPassword	JDBCOracleParams.setOnsWalletPassword()
OracleOptimizeUtf8Conversion	JDBCOracleParams.setOracleOptimizeUtf8Conversion()
PasswordEncrypted	JDBCDriverParams.setPassword()
PinnedToThread	JDBCConnectionPoolParams.setPinnedToThread()
ProfileHarvestFrequencySeconds	JDBCConnectionPoolParams.setProfileHarvestFrequencySeconds()
ProfileType	JDBCConnectionPoolParams.setProfileType()
RecoverOnlyOnce	JDBCXAParams.setRecoverOnlyOnce()
RemoveInfectedConnections	JDBCConnectionPoolParams.setRemoveInfectedConnections()
ResourceHealthMonitoring	JDBCXAParams.setResourceHealthMonitoring()
RollbackLocalTxUponConnClose	JDBCXAParams.setRollbackLocalTxUponConnClose()
RowPrefetch	JDBCDataSourceParams.setRowPrefetch()
RowPrefetchSize	JDBCDataSourceParams.setRowPrefetchSize()
SecondsToTrustAnIdlePoolConnection	JDBCConnectionPoolParams.setSecondsToTrustAnIdlePoolConnection()
ShrinkFrequencySeconds	JDBCConnectionPoolParams.setShrinkFrequencySeconds()
StatementCacheSize	JDBCConnectionPoolParams.setStatementCacheSize()
StatementCacheType	JDBCConnectionPoolParams.setStatementCacheType()
StatementTimeout	JDBCConnectionPoolParams.setStatementTimeout()
StreamChunkSize	JDBCDataSourceParams.setStreamChunkSize()
TestConnectionsOnReserve	JDBCConnectionPoolParams.setTestConnectionsOnReserve()
TestFrequencySeconds	JDBCConnectionPoolParams.setTestFrequencySeconds()
TestTableName	JDBCConnectionPoolParams.setTestTableName()
UsePasswordIndirection	JDBCDriverParams.setUsePasswordIndirection()

Table 3-1 (Cont.) WebLogic Configuration Attributes

Weblogic.Attribute Name	WebLogic Element
UseXaDataSourceInterface	JDBCDriverParams.setUseXaDataSourceInterface()
WrapTypes	JDBCConnectionPoolParams.setWrapTypes()
XaEndOnlyOnce	JDBCXAParams.setXaEndOnlyOnce()
XaRetryDurationSeconds	JDBCXAParams.setXaRetryDurationSeconds()
XaRetryIntervalSeconds	JDBCXAParams.setXaRetryIntervalSeconds()
XaSetTransactionTimeout	JDBCXAParams.setXaSetTransactionTimeout()
XaTransactionTimeout	JDBCXAParams.setXaTransactionTimeout()

Implementation Considerations When Using DataSource Resource Definitions

Learn about the implementation details to consider when creating and using DataSource resource definitions.

- [Naming Conventions](#)
- [Mapping the Java EE DataSource Resource Definition to WebLogic Data Source Resources](#)
- [Configuring Active GridLink DataSource Resource Definitions](#)
- [Using an Encrypted Password in a DataSourceDefinition](#)
- [Additional Considerations](#)

Naming Conventions

This section provides information on Data Source naming conventions:



Note:

Pre-WebLogic Server 12.1.1 and Java EE Data Source naming conventions are compatible. Existing applications do not need to change naming conventions to upgrade from previous releases.

WebLogic Data Source Naming Conventions

The following conventions are used when naming Data Sources in releases prior to WebLogic Server 12.1.1:

- *dsname* - The system resource JDBC descriptor (`config/jdbc/*-jdbc.xml`)

- *application@null@dsname* - deprecated (pre-9.x), application-scoped JDBC descriptor in EAR
- *application@module@dsname* - application-scoped, packaged JDBC descriptor in EAR

Java EE Data Source Naming Conventions

The following conventions are used to name Java EE Data Sources:

- *appName@moduleName@componentname@dsname* - Component level
- *appName@moduleName@dsname* - Module level
- *appName@dsname* - Application level
- *dsname* – Global

These names are compatible with earlier names because the Java EE names begin with *java*:

Mapping the Java EE DataSource Resource Definition to WebLogic Data Source Resources

[Table 3-2](#) provides information on how to map Java EE DataSource Resource definition elements to WebLogic Server resources.

Table 3-2 Mapping a DataSource Resource Definition to WebLogic Server Resources

DataSourceBean	Default Value	WebLogic Resource
String name()	Required	JDBCDataSourceParamsBean.setJndiName
String className()	Required	JDBCDriverParamsBean.setDriverName
String description()	""	Not Used
String url()	""	JDBCDriverParamsBean.setUrl
String user()	""	Added to JDBCDriverParamsBean.getProperties()
String password()	""	JDBCDriverParamsBean.setPassword
String databaseName()	""	Used to generate URL; added to properties
int portNumber()	-1	Used to generate URL; added to properties
String serverName()	"localhost"	Used to generate URL; added to properties

Table 3-2 (Cont.) Mapping a DataSource Resource Definition to WebLogic Server Resources

DataSourceBean	Default Value	WebLogic Resource
int isolationLevel()	-1	Sets desiredtxisolevel property which WebLogic Server uses to call Connection.setTransactionIsolation()
boolean transactional()	true	Used to generate URL
int initialPoolSize()	-1	JDBCConnectionPoolParams Bean.setInitialCapacity
int maxPoolSize()	-1	JDBCConnectionPoolParams Bean.setMaxCapacity
int minPoolSize()	-1	JDBCConnectionPoolParams Bean.setMinCapacity (new)
int maxIdleTime()	-1	JDBCConnectionPoolParams Bean.setShrinkFrequencySeconds
int maxStatements()	-1	JDBCConnectionPoolParams Bean.setStatementCacheSize
String[] properties()	{}	JDBCPropertiesBean
int loginTimeout()	0	Not Used

Configuring Active GridLink DataSource Resource Definitions

An Active GridLink Data Source is defined by using the following name/value pair within the DataSource resource definition:

- FanEnabled is set to true
- OnsNodeList is a non-null value. A comma-separated list of ONS daemon listen addresses and ports for receiving ONS-based FAN events. See ONS Client Configuration in *Administering JDBC Data Sources for Oracle WebLogic Server*.

The following example shows a DataSource resource definition for an Active GridLink Data Source using deployment descriptors:

```

. . .

<data-source>
  <name>java:global/DSD2</name>
  <class-name>oracle.jdbc.OracleDriver</class-name>
  <url>jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
(HOST=lcr01155-r)(PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=mydb))</url>
  <user>lefty123</user>
  <password>password</password>
  <property><name>weblogic.CapacityIncrement</name><value>2</value></property>
  <property><name>weblogic.HighestNumWaiters</name><value>2147483647</value></
property>

```

```

    <property><name>weblogic.ConnectionCreationRetryFrequencySeconds</
name><value>0</value></property>
    <property><name>weblogic.ConnectionReserveTimeoutSeconds</name><value>10</
value></property>
    <property><name>weblogic.TestFrequencySeconds</name><value>120</value></
property>
    <property><name>weblogic.TestConnectionsOnReserve</name><value>false</
value></property>
    <property><name>weblogic.ProfileHarvestFrequencySeconds</name><value>300</
value></property>
    <property><name>weblogic.IgnoreInUseConnectionsEnabled</name><value>true</
value></property>
    <property><name>weblogic.InactiveConnectionTimeoutSeconds</name><value>0</
value></property>
    <property><name>weblogic.TestTableName</name><value></value></property>
    <property><name>weblogic.LoginDelaySeconds</name><value>0</value></property>
    <property><name>weblogic.InitSql</name><value></value></property>
    <property><name>weblogic.StatementCacheType</name><value>LRU</value></
property>
    <property><name>weblogic.RemoveInfectedConnections</name><value>true</
value></property>
    <property><name>weblogic.SecondsToTrustAnIdlePoolConnection</name><value>10</
value></property>
    <property><name>weblogic.StatementTimeout</name><value>-1</value></property>
    <property><name>weblogic.ProfileType</name><value>0</value></property>
    <property><name>weblogic.JDBCXADebugLevel</name><value>10</value></property>
    <property><name>weblogic.CredentialMappingEnabled</name><value>false</
value></property>
    <property><name>weblogic.DriverInterceptor</name><value></value></property>
    <property><name>weblogic.PinnedToThread</name><value>false</value></property>
    <property><name>weblogic.IdentityBasedConnectionPoolingEnabled</
name><value>false</value></property>
    <property><name>weblogic.WrapTypes</name><value>true</value></property>
    <property><name>weblogic.ConnectionLabelingCallback</name><value></value></
property>
    <property><name>weblogic.FatalErrorCodes</name><value></value></property>
    <property><name>weblogic.Scope</name><value>Global</value></property>
    <property><name>weblogic.RowPrefetch</name><value>false</value></property>
    <property><name>weblogic.RowPrefetchSize</name><value>48</value></property>
    <property><name>weblogic.StreamChunkSize</name><value>256</value></property>
    <property><name>weblogic.AlgorithmType</name><value>Failover</value></
property>
    <property><name>weblogic.ConnectionPoolFailoverCallbackHandler</
name><value></value></property>
    <property><name>weblogic.FailoverRequestIfBusy</name><value>false</value></
property>
    <property><name>weblogic.GlobalTransactionsProtocol</
name><value>OnePhaseCommit</value></property>
    <property><name>weblogic.KeepConnAfterLocalTx</name><value>true</value></
property>
    <property><name>weblogic.KeepConnAfterGlobalTx</name><value>false</value></
property>
    <property><name>weblogic.UseXaDataSourceInterface</name><value>true</value></
property>
    <property><name>weblogic.UsePasswordIndirection</name><value>false</value></
property>
    <property><name>weblogic.FanEnabled</name><value>true</value></property>
    <property><name>weblogic.OnsNodeList</name><value>lcr01155-r:6200</value></
property>
    <property><name>weblogic.OnsWalletFile</name><value></value></property>
    <property><name>weblogic.OnsWalletPassword</name><value></value></property>

```



```

    <property><name>weblogic.OracleOptimizeUtf8Conversion</name><value>>false</
value></property>
    <property><name>weblogic.ConnectionInitializationCallback</name><value></
value></property>
    <property><name>weblogic.AffinityPolicy</name><value>Session</value></
property>
    <property><name>weblogic.OracleProxySession</name><value>>false</value></
property>
    <property><name>weblogic.KeepXaConnTillTxComplete</name><value>>true</value></
property>
    <property><name>weblogic.NeedTxCtxOnClose</name><value>>false</value></
property>
    <property><name>weblogic.XaEndOnlyOnce</name><value>>false</value></property>
    <property><name>weblogic.NewXaConnForCommit</name><value>>false</value></
property>
    <property><name>weblogic.KeepLogicalConnOpenOnRelease</name><value>>false</
value></property>
    <property><name>weblogic.ResourceHealthMonitoring</name><value>>true</value></
property>
    <property><name>weblogic.RecoverOnlyOnce</name><value>>false</value></
property>
    <property><name>weblogic.XaSetTransactionTimeout</name><value>>false</value></
property>
    <property><name>weblogic.XaTransactionTimeout</name><value>0</value></
property>
    <property><name>weblogic.RollbackLocalTxUponConnClose</name><value>>false</
value></property>
    <property><name>weblogic.XaRetryDurationSeconds</name><value>0</value></
property>
    <property><name>weblogic.XaRetryIntervalSeconds</name><value>60</value></
property>
  </data-source>
  . . .

```

For additional information, see Using Active GridLink Data Sources in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Using an Encrypted Password in a DataSourceDefinition

You can provide an encrypted password in the `DataSourceDefinition`. To do so you need to generate the password as shown in the following example, and then copy it into the `DataSourceDefinition`:

```

# needs to be run in the domain home directory
java weblogic.security.Encrypt
Password: user_password
{AES}OQ1CnXWsgTVQsrxHqpxMT7iZwt7wBBIrkLP5NWeAvNk="
# This value needs to be pasted into the DataSourceDefinition

```

The encrypted password is domain specific. If you use an encrypted password that does not match the domain, it will generate an error such as:

```

weblogic.application.ModuleException: com.rsa.jsafe.JSAFE_PaddingException:
  Invalid padding.:com.rsa.jsafe.JSAFE_PaddingException:Invalid padding

```

The following code fragment defines a data source using an encrypted password in an annotation in a Java servlet.

```

@DataSourceDefinition(
name="java:comp/ds",

```

```
className="oracle.jdbc.OracleDriver",
portNumber=1521,
serverName="myhost",
user="myuser",
databaseName="mydbname",
initialPoolSize = 0,
minPoolSize = 0,
maxPoolSize = 15,
maxStatements = 0,
transactional=false,
properties = {"weblogic.TestTableName=SQL ISVALID",
  "weblogic.PasswordEncrypted={AES}OQ1CnXWsgTVQsxrHqpxMT7iZwt7wBBIrkLP5NWeAvNk=" }
)
@WebServlet(urlPatterns = "/GetVersion")
public class GetVersion extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {
  @Resource(lookup = "java:comp/ds")
  private DataSource ds;
```

Additional Considerations

Consider the following when using a Java EE `DataSource` resource definition with WebLogic Server:

- If an annotation and a descriptor have the same `DataSource` name in the same scope, the attributes are merged with values specified in the deployment descriptor. The deployment descriptor value takes precedence over values specified in a annotation.
- A `DataSource` is not a module, it is a resource created as part of a module.
- A `DataSource` is not a `JDBCSystemResources` object associated with a domain and is not in the WebLogic Server configuration bean tree.
- You can use the JSR88 API's to view applications that include Java EE 7 Data Sources.
- There is one runtime MBean created for each `datasource` definition. The name of the MBean is the decorated name.
- WLS has a limited set of known class names for which it can generate a URL. For the non-XA case, the JDBC driver and not the `datasource` class is often known. An error occurs when an unknown class name is specified with a `databaseName`, `portNumber`, and/or `serverName`. In this case, remove `databaseName`, `portNumber`, `serverName`, and specify the URL.
- URL generation is not supported for AGL data sources.
- URL generation in general is a problem for all Oracle drivers because of the service, database, and Oracle RAC instance formats. The best practice is to provide the URL for Oracle drivers

Using Data Sources in Clients

WebLogic Server allows you to implement Java EE data sources in a Java EE client with some limitations.

The limitations are as follows:

- `Transactional=true` is not supported. The transaction protocol is set to `NONE`.

- Data Sources that are global or application in scope are visible (created) both on the client and the server. This has the downside of using more connections.
- No permission checking is performed on a Data Source. Operations such as `reserve` and `shrink` can be used on a local Data Source.

Additional Resources

Learn about additional resources for review when implementing data source resource definitions.

- Java EE 7 Specification at <https://www.jcp.org/en/jsr/detail?id=342>
- The Java EE 7 Tutorial at <http://docs.oracle.com/javaee/7/tutorial/doc/>
- JDBC™ 4.1 Specification at http://download.oracle.com/otn-pub/jcp/jdbc-4_1-mrel-spec/jdbc4.1-fr-spec.pdf
- Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*

4

Performance Tuning Your JDBC Application

Learn how to design and configure WebLogic Server to get the best performance from JDBC applications.

- [WebLogic Performance-Enhancing Features](#)
- [Designing Your Application for Best Performance](#)

WebLogic Performance-Enhancing Features

WebLogic has several features that enhance performance for JDBC applications including pooled connections and caching statements.

How Pooled Connections Enhance Performance

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

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

Caching Statements and Data

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

- Statement Cache for a data source
- Read-Only Entity Beans
- JNDI in a Clustered Environment

Designing Your Application for Best Performance

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

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

Process as Much Data as Possible Inside the Database

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

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

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

Use Built-in DBMS Set-based Processing

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

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

Make Your Queries Smart

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

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

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

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

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

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

Proper use of SQL can avoid these performance problems. For example, if you only want data about the top three earners on the payroll, the proper way to make this query is with a correlated subquery. [Table 4-1](#) shows the entire table returned by the SQL statement

```
select * from payroll
```

Table 4-1 Full Results Returned

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

A correlated subquery

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

returns a much smaller result, shown in [Table 4-2](#).

Table 4-2 Results from Subquery

Name	Salary
Sue	70
Hal	80
May	80

This query returns only *three rows, with the name and salary of the top three earners*. It scans through the payroll table, and for every row, it goes through the whole payroll table again in an inner loop to see how many salaries are higher than the current row

of the outer scan. This may look complicated, but DBMSs are designed to use SQL efficiently for this type of operation.

Make Transactions Single-batch

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

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

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

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

Never Have a DBMS Transaction Span User Input

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

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

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

Use In-place Updates

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

Keep Operational Data Sets Small

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

Use Pipelining and Parallelism

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

5

Using WebLogic-branded DataDirect Drivers

Learn about the WebLogic-branded DataDirect drivers that are included in the WebLogic Server distribution.

- [Using DataDirect Documentation](#)
- [JDBC Specification Compliance](#)
- [Installation](#)
- [Supported Drivers and Databases](#)
- [Connecting Through WebLogic JDBC Data Sources](#)
- [Developing Your Own JDBC Code](#)
- [Specifying Connection Properties](#)
- [Using IP Addresses](#)
- [Required Permissions for the Java Security Manager](#)
- [For MS SQLServer Users](#)

Using DataDirect Documentation

Oracle provides WebLogic-branded versions of DataDirect drivers for DB2, Informix, MS SQL Server, and Sybase. Learn how WebLogic-branded DataDirect drivers are configured and used in a WebLogic Server environment.

For detailed information on these drivers, see *Progress DataDirect Connect Series for JDBC User's Guide Release 5.1* and *Progress DataDirect for JDBC Drivers Reference Release 5.1* at [DataDirect Connectors for JDBC Documentation](#). You need to make the following adaptations where appropriate when using DataDirect documentation:

- URLs: substitute "weblogic" for "datadirect"
- Install directory: the fully qualified installation directory for WebLogic-branded DataDirect drivers is `ORACLE_HOME\oracle_common\modules\datadirect`.

JDBC Specification Compliance

WebLogic-branded Data Direct drivers are compliant with the JDBC 4.0 specification.

 **Note:**

When comparing WebLogic Server behavior when using drivers from different vendors, it is important to remember that even though the drivers are JDBC specification compliant, a vendor may interpret the specification differently or provide different implementations for a given situation.

For example: When using the WebLogic-branded SQL Server driver, if you enter a negative value (-100) into a `TINYINT` column where the schema defines the range as 0 to 256, the driver throws an exception, whereas the Microsoft SQL Server driver ignores the minus sign.

Installation

Learn about the installation of DataDirect drivers with WebLogic Server.

WebLogic-branded DataDirect drivers are installed with WebLogic Server in the `ORACLE_HOME\oracle_common\modules\datadirect` folder, where `ORACLE_HOME` is the directory in which you installed WebLogic Server. Driver jar files are included in the manifest classpath in `weblogic.jar`, so the drivers are automatically added to your classpath on the server.

 **Note:**

WebLogic-branded DataDirect drivers are installed by default when you perform a complete installation of WebLogic Server. If you choose a custom installation, ensure that the WebLogic JDBC Drivers option is selected (checked). If this option is unchecked, the drivers are not installed.

WebLogic-branded DataDirect drivers are not included in the manifest classpath of the WebLogic client jar files (for example: `wlclient.jar`). To use the drivers with a WebLogic client, you must copy the following files to the client and add them to the classpath on the client:

- For DB2: `wldb2.jar`
- For Informix: `wlinformix.jar`
- For MS SQL Server: `wlsqlserver.jar`
- For Sybase: `wlsybase.jar`

Supported Drivers and Databases

Learn about supported drivers and databases.

For information on driver and database support, see <http://www.oracle.com/technetwork/middleware/ias/downloads/fusion-certification-100350.html>.

Connecting Through WebLogic JDBC Data Sources

To create a physical database connection in the data source, create a JDBC data source in your WebLogic Server configuration and select the JDBC driver.

.Applications can then look up the data source on the JNDI tree and request a connection.

See the following related information:

- For information about JDBC and data sources in WebLogic Server, see *Configuring JDBC Data Sources in Administering JDBC Data Sources for Oracle WebLogic Server*.
- For information about requesting a connection from a data source, see [Obtaining a Client Connection Using a DataSource](#).

Developing Your Own JDBC Code

You can develop JDBC code that uses the WebLogic-branded DataDirect drivers as long as the code is included in the WebLogic Server CLASSPATH.

Specifying Connection Properties

You specify connection properties for connections in a data source using the WebLogic Server Administration Console, command-line interface, or JMX API. Connection properties vary by DBMS.

For the list of the connection properties specific to each of the WebLogic-branded DataDirect drivers, see the **Connection Properties** section for your driver in [Progress DataDirect for JDBC User's Guide](#).

Using IP Addresses

WebLogic-branded DataDirect drivers support Internet Protocol (IP) addresses in IPv4 and IPv6 format.

See [Progress DataDirect for JDBC User's Guide Release 5.1](#) for more details. In a WebLogic environment, simply convert the `jdbc:datadirect` portion of the URL to `jdbc:weblogic`. For example, the following connection URL specifies the server using IPv4 format:

```
jdbc:weblogic:db2://123.456.78.90:50000;DatabaseName=jdbc;User=test;  
Password=secret
```

Required Permissions for the Java Security Manager

Using WebLogic-branded DataDirect drivers with the Java Security Manager enabled requires certain permissions to be set in the security policy file of the domain. WebLogic Server provides a sample security policy file that you can edit and use.

The file is located at `ORACLE_HOME\wlserver\server\lib`. The `weblogic.policy` file includes all necessary permissions for the drivers.

If you use the `weblogic.policy` file without changes, you may not need to grant any further permissions. If you use another security policy file or if you use driver features that require additional permissions, see [Progress DataDirect for JDBC User's Guide Release 5.1](#) for details. Use `ORACLE_HOME\oracle_common\modules\datadirect` as the `install_dir` where `ORACLE_HOME` is the directory in which you installed WebLogic Server.

For more information about using the Java Security Manager with WebLogic Server, see Using Java Security to Protect WebLogic Resources in *Developing Applications with the WebLogic Security Service*.

For MS SQLServer Users

Learn about configuring MS SQLServer for use with DataDirect MS SQL Server driver.

- [Installing MS SQLServer XA DLLs](#)
- [Using `instjdbc.sql` with MS SQLServer](#)

Installing MS SQLServer XA DLLs

WebLogic Server provides the following XA dlls for MS SQL Server:

- `sqljdbc.dll`: for 32-bit Windows
- `64sqljdbc.dll`: for 64-bit Windows
- `X64sqljdbc.dll`: for the X64 processors

To install, do the following:

1. `cd` to the `ORACLE_HOME\oracle_common\modules\datadirect` directory
2. For:
 - 32-bit Windows systems, install the `sqljdbc.dll` file.
 - 64-bit Windows systems, copy the `64sqljdbc.dll` file, rename as `sqljdbc.dll`, and then install the `sqljdbc.dll` file.
 - X64 processors, copy the `X64sqljdbc.dll` file, rename as `sqljdbc.dll`, and then install the `sqljdbc.dll` file.

Using `instjdbc.sql` with MS SQLServer

There is a known error in some versions of the DataDirect `instjdbc.sql` script that installs stored procedures into MS SQLServer versions 2008 and newer. The workaround is to replace all instances of `dump tran master` with `no_log` in the `instjdbc.sql` script with `DBCC SHRINKFILE(mastlog, 1)`.

6

Using WebLogic Wrapper Drivers

Learn how to use deprecated WebLogic wrapper drivers with WebLogic Server.

Note:

Oracle recommends that you use `DataSource` objects to get database connections in new applications. `DataSource` objects, along with the JNDI tree, provide access to pooled connections in a data source for database connectivity. The WebLogic wrapper drivers are deprecated. For existing or legacy applications that use the JDBC 1.x API, you can use the WebLogic wrapper drivers to get database connectivity.

This chapter includes the following sections:

- [Using the WebLogic RMI Driver \(Deprecated\)](#)
- [Using the WebLogic JTS Driver \(Deprecated\)](#)

Using the WebLogic RMI Driver (Deprecated)

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

Note:

RMI driver client functionality is deprecated and will be removed in future release. None of the features exposed in `WLConnection` and `WLDataSource` are supported by RMI driver clients.

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

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

RMI Driver Client Interoperability

Interoperability with earlier WebLogic Server releases is limited. Participants (client/server or servers-to-server) must be from the same major release. Early 10.x clients

can be updated to interoperate with later point and patch set releases by adding the `ucp.jar` to the `CLASSPATH`.

Security Considerations for WebLogic RMI Drivers

Applications that use JDBC over RMI allow unauthorized RMI access to a `DataSource` object, which is a potential security vulnerability as it can provide a client with uncontrolled access to a database. Oracle recommends replacing JDBC over RMI with local WebLogic data sources in these environments.

You can control JDBC over RMI communications by setting the `RmiJDBCSecurity` parameter in the `DataSource` object at the server level.

See [Enable RMI JDBC security](#) in the *Oracle WebLogic Server Administration Console Online Help*.

The following are the valid values of the parameter:

- `Compatibility` - Allows uncontrolled access to `DataSource` objects for all incoming JDBC application calls over RMI. This setting should only be used when strong network security is in place. `Compatibility` is the default value.
- `Secure` - Rejects all incoming application JDBC calls over RMI by remote clients and servers. Internal interserver JDBC calls over RMI operations are allowed for the Logging Last Resource, Emulate Two-Phase Commit and One-Phase Commit Global Transactions Protocol options. The `Secure` option requires that all the servers are configured with an SSL listen port. If not, all operations fail with an exception.

Note:

The `weblogic.jdbc.remoteEnabled` (deprecated) system property is enabled by default. You can completely disable RMI access to `DataSource` objects by setting the `weblogic.jdbc.remoteEnabled` system property to `false`.

Setting Up WebLogic Server to Use the WebLogic RMI Driver

The RMI driver is accessible through `DataSource` objects, which are created in the WebLogic Server Administration Console. You should create `DataSource` objects in your WebLogic Server configuration before you use the RMI driver in your applications.

Sample Client Code for Using the RMI Driver

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

Import the Required Packages

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

```
javax.sql.DataSource  
java.sql.*  
java.util.*  
javax.naming.*
```

Get the Database Connection

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

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

Using a JNDI Lookup to Obtain the Connection

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

```
Context ctx = null;  
    Hashtable ht = new Hashtable();  
    ht.put(Context.INITIAL_CONTEXT_FACTORY,  
           "weblogic.jndi.WLInitialContextFactory");  
    ht.put(Context.PROVIDER_URL,  
           "t3://hostname:port");  
    try {  
        ctx = new InitialContext(ht);  
        javax.sql.DataSource ds  
            = (javax.sql.DataSource) ctx.lookup ("myDataSource");  
        java.sql.Connection conn = ds.getConnection();  
        // You can now use the conn object to create  
        // a Statement object to execute  
        // SQL statements and process result sets:  
        Statement stmt = conn.createStatement();  
        stmt.execute("select * from someTable");  
        ResultSet rs = stmt.getResultSet();  
        // Do not forget to close the statement and connection objects  
        // when you are finished:  
    }  
    catch (Exception e) {  
        // a failure occurred  
        log message;  
    }  
} finally {  
    try {  
        ctx.close();  
    } catch (Exception e) {  
        log message; }  
    try {  
        if (rs != null) rs.close();  
    } catch (Exception e) {  
        log message; }  
    try {  
        if (stmt != null) stmt.close();  
    } catch (Exception e) {  
        log message; }  
}
```

```
try {
    if (conn != null) conn.close();
} catch (Exception e) {
    log message; }
}
```

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

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. See WebLogic Server JNDI in *Developing JNDI Applications for Oracle WebLogic Server*.

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

 **Note:**

It may be possible to access a vendor-specific interface. This is done without RMI by casting to the vendor interface. For example:

```
OracleConnection oc = (OracleConnection) cconn;
```

This may not work if the vendor interface is not *Serializable*. When a server is acting as a client, set *networkClassLoadingEnabled* to *true* on the server so that the generated RMI class is available (the default is *true* for stand-alone clients).

Using Only the WebLogic RMI Driver to Obtain a Database Connection

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

```
java.sql.Driver myDriver = (java.sql.Driver)
    Class.forName("weblogic.jdbc.rmi.Driver").newInstance();
String url = "jdbc:weblogic:rmi";
java.util.Properties props = new java.util.Properties();
props.put("weblogic.server.url", "t3://hostname:port");
props.put("weblogic.jdbc.datasource", "myDataSource");
java.sql.Connection conn = myDriver.connect(url, props);
```

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

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

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

Row Caching with the WebLogic RMI Driver

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

 **Note:**

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

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

1. If you have not already done so, in the Change Center of the WebLogic Server Administration Console, click **Lock & Edit**.
2. In the Domain Structure tree, expand Services > JDBC, then select Data Sources.
3. On the Summary of Data Sources page, click the data source name.
4. Select the Configuration: General tab and then do the following:
 - a. Select the Row Prefetch Enabled check box.
 - b. In Row Prefetch Size, type the number of rows you want to cache for each `ResultSet.next()` call.
5. Click **Save**.
6. To activate these changes, in the Change Center of the WebLogic Server Administration Console, click Activate Changes.

See the [JDBC Data Source: Configuration: General](#) page in the *Oracle WebLogic Server Administration Console Online Help*.

Important Limitations for Row Caching with the WebLogic RMI Driver

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

- WebLogic Server only performs row caching if the result set type is both `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY`.
- Certain data types in a result set may disable caching for that result set. These include the following:
 - `LONGVARCHAR/LONGVARBINARY`
 - `NULL`

- BLOB/CLOB
 - ARRAY
 - REF
 - STRUCT
 - JAVA_OBJECT
- Certain `ResultSet` methods are not supported if row caching is enabled and active for that result set. Most pertain to streaming data, scrollable result sets or data types not supported for row caching. These include the following:
 - `getAsciiStream()`
 - `getUnicodeStream()`
 - `getBinaryStream()`
 - `getCharacterStream()`
 - `isBeforeLast()`
 - `isAfterLast()`
 - `isFirst()`
 - `isLast()`
 - `getRow()`
 - `getObject (Map)`
 - `getRef()`
 - `getBlob()/getClob()`
 - `getArray()`
 - `getDate()`
 - `getTime()`
 - `getTimestamp()`

Limitations When Using Global Transactions

Populating a `RowSet` in a global transaction may fail with `Fetch Out Of Sequence` exception. For example:

1. When the RMI call returns, the global transaction is suspended automatically by the server instance.
2. The JDBC driver invalidates the pending `ResultSet` object to release the system resources.
3. The client tries to read data from the invalidated `ResultSet`.
4. A `Fetch Out Of Sequence` exception is thrown if that data has not been prefetched. Since the number of rows prefetched is vendor specific, you may or may not encounter this issue, especially when working with one or two rows.

If you encounter this exception, make sure to populate the `RowSet` on the server side and then serialize it back to the client.

Using the WebLogic JTS Driver (Deprecated)

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

Your application uses the JTS driver to access a connection from the data source.

WebLogic Server also uses the JTS driver internally when a connection from a data source that uses a non-XA JDBC driver participates in a global transaction (Logging Last Resource and Emulate Two-Phase Commit). This behavior enables a non-XA resource to emulate XA and participate in a two-phase commit transaction. See JDBC Data Source Transaction Options in *Administering JDBC Data Sources for Oracle WebLogic Server*.

 **Note:**

The WebLogic Server JTS driver only supports T3 protocol when participating connections that use Logging Last Resource (LLR).

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

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

For the JTS driver and your application to participate in a global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` within a global transaction. After the transaction completes (gets committed or rolled back), WebLogic Server puts the connection back in the data source. If you want to use a connection for another global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` again within a new global transaction.

Sample Client Code for Using the JTS Driver

To use the JTS driver, you must first use the WebLogic Server Administration Console to create a data source in WebLogic Server.

This explanation demonstrates creating and using a JTS transaction from a server-side application and uses a data source named "myDataSource."

1. Import the following classes:

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

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

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

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

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

ctx = new InitialContext(env);

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

3. Start a transaction on the current thread:

```
// Start the global transaction before getting a connection
tx.begin();
```

4. Load the JTS driver:

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

5. Get a connection from the data source:

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

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

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

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

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

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

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

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

```
// or:
```

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

7

Using API Extensions in JDBC Drivers

Learn how to configure and use third-party JDBC drivers, including using API extensions and batch processing, with Oracle Thin Drivers.

- [Using API Extensions to JDBC Interfaces](#)
- [Using API Extensions for Oracle JDBC Types](#)
- [Using Batching with the Oracle Thin Driver](#)
- [Using the Java Security Manager with the Oracle Thin Driver](#)

Using API Extensions to JDBC Interfaces

WebLogic Server has implemented new interfaces for Oracle JDBC Types. Learn about the new interfaces and how they map to the deprecated Oracle concrete classes.

To use the extension methods exposed in the JDBC driver, you must include these steps in your application code:

- Import the driver interfaces from the JDBC driver used to create connections in the data source.
- Get a connection from the data source.
- Cast the connection object as the vendor's connection interface.
- Use the API extensions as described in the vendor's documentation.
- Wrap the JNDI lookup in a `try/catch` block in order to catch a failed look up and ensure the context is closed in a `finally` block.

The following sections provide details on using API extensions and supporting code examples. For information about specific extension methods for a particular JDBC driver, refer to the documentation from the JDBC driver vendor.

Sample Code for Accessing API Extensions to JDBC Interfaces

The following code examples use extension methods available in the Oracle Thin driver to illustrate how to use API extensions to JDBC. You can adapt these examples to fit methods exposed in your JDBC driver.

Import Packages to Access API Extensions

Import the interfaces from the JDBC driver used to create the connection in the data source. This example uses interfaces from the Oracle Thin Driver.

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import oracle.jdbc.*;
```

```
// Import driver interfaces. The driver must be the same driver
// used to create the database connection in the data source.
```

Get a Connection

Establish the database connection using JNDI, `DataSource`, and data source objects.

```
// Get a valid DataSource object for a data source.
// Here we assume that getDataSource() takes
// care of those details.
javax.sql.DataSource ds = getDataSource(args);
// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Cast the Connection as a Vendor Connection

Now that you have the connection, you can cast it as a vendor connection. This example uses the `OracleConnection` interface from the Oracle Thin Driver.

```
OracleConnection = (oracle.jdbc.OracleConnection)conn;
```

Use API Extensions

The following code fragment shows how to use the Oracle `Row Prefetch` method available from the Oracle Thin driver.

Example 7-1 Using an API Extension

```
// Cast to OracleConnection and retrieve the
// default row prefetch value for this connection.
int default_prefetch =
    ((oracle.jdbc.OracleConnection)conn).getDefaultRowPrefetch();
// Cast to OracleStatement and set the row prefetch
// value for this statement. Note that this
// prefetch value applies to the connection between
// WebLogic Server and the database.
    ((oracle.jdbc.OracleStatement)stmt).setRowPrefetch(20);
// Perform a normal sql query and process the results...
String query = "select empno,ename from emp";
java.sql.ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
    java.math.BigDecimal empno = rs.getBigDecimal(1);
    String ename = rs.getString(2);
    System.out.println(empno + "\t" + ename);
}
rs.close();
stmt.close();
conn.close();
conn = null;
. . .
```

Using API Extensions for Oracle JDBC Types

WebLogic Server has implemented new interfaces for Oracle JDBC Types. Learn about the new interfaces and how they map to the deprecated Oracle concrete classes.

When Oracle implemented JDBC, concrete classes were used instead of using interfaces for Oracle JDBC Types. There are many of drawbacks in using concrete

classes and in the 11.2.0.3 driver there are new interfaces corresponding to the Oracle types. The concrete classes now implement a public interface from the package `oracle.jdbc`. Programmers should use methods exposed in `java.sql` whenever possible and for Oracle extension methods use `oracle.jdbc`.

In the mean time, WebLogic Server implemented corresponding interfaces that could be used to work around the limitations of the concrete classes. These are now deprecated and should be replaced with the corresponding `oracle.jdbc` interfaces.

In Database version 11.2.0.3 the following types have interfaces.

Old Oracle types	Deprecated WLS Interface	New interfaces
<code>oracle.sql.ARRAY</code>	<code>weblogic.jdbc.vendor.oracle.OracleArray</code>	<code>oracle.jdbc.OracleArray</code>
<code>oracle.sql.STRUCT</code>	<code>weblogic.jdbc.vendor.oracle.OracleStruct</code>	<code>oracle.jdbc.OracleStruct</code>
<code>oracle.sql.CLOB</code>	<code>weblogic.jdbc.vendor.oracle.OracleThinClob</code>	<code>oracle.jdbc.OracleClob</code>
<code>oracle.sql.BLOB</code>	<code>weblogic.jdbc.vendor.oracle.OracleThinBlob</code>	<code>oracle.jdbc.OracleBlob</code>
<code>oracle.sql.REF</code>	<code>weblogic.jdbc.vendor.oracle.OracleRef</code>	<code>oracle.jdbc.OracleRef</code>

Changing the code to use new interfaces is not difficult, but should be handled with care. The below examples use `oracle.sql.ARRAY` and similar changes apply to other types as well. A list of suggested changes is mentioned below:

- **Import:** Modify import statements to use the new interfaces (`oracle.jdbc`) instead of old interfaces (`oracle.sql` or `weblogic.jdbc.vendor.oracle`).
- **Declaration:** Use standard Java interfaces for declaration whenever possible. If there is a need to use Oracle extension, use the new Oracle interfaces under `oracle.jdbc`.
- **Methods:** Use standard Java interfaces whenever possible:
 - (Oracle Types): Use methods in standard Java interfaces whenever possible. If required use methods from Oracle interfaces under `oracle.jdbc`.
 - (Defines): Refrain from using Oracle specific methods such as `getARRAY`; instead use standard Java methods such as `getArray` or `getObject` for those that do have standard Java interfaces.
 - (Binds): Refrain from using Oracle specific methods such as `setARRAY`; instead use standard Java methods such as `setArray` or `setObject` for the ones that do have standard Java interfaces.

Replacing import statements can be done by a script that uses `find` and `sed`. For example:

```
find . -name "*.java" -exec egrep ... > files.list

for f in `cat files.list`; do

    cat $f |sed 's@^import oracle\.sql\.ARRAY@oracle\.jdbc\.OracleArray@g' > /tmp/
temp.txt

    mv /tmp/temp.txt $f
```



```
done
```

Programmers should use factory methods on `oracle.jdbc.OracleConnection` to create an instance of the types. For example:

```
int[] intArray = { 5, 7, 9};

oracle.sql.ArrayDescriptor aDescriptor = new
oracle.sql.ArrayDescriptor("SCOTT.TYPE1", connection);

oracle.sql.ARRAY array = new oracle.sql.ARRAY(aDescriptor, connection, intArray);
```

should be changed to:

```
int[] intArray = { 5, 7, 9};
java.sql.Array array = connection.createOracleArray("SCOTT.TYPE1", intArray);
```

 **Note:**

Oracle does not support anonymous array types and so does not support the standard `Connection.createArrayOf` method. Instead, use `createOracleArray` as shown in the sample above.

There are some methods that are no longer available because:

- There is a way to accomplish the same end using standard or already public methods.
- The method refers to a deprecated type.
- The method does not add significant value.

In these cases, the code needs to be modified to use standard API's.

Sample Code for Accessing Oracle Thin Driver Extensions to JDBC Interfaces

The following code examples show how to access the interfaces for Oracle extensions, including interfaces for:

- Arrays—See [Programming with Arrays](#).
- Structs—See [Programming with Structs](#).
- Refs—See [Programming with Refs](#).
- Blobs and Clobs—See [Programming with Large Objects](#).

If you selected the option to install server examples with WebLogic Server, see the JDBC examples for more code examples, see [JDBC Samples and Tutorials](#).

 **Note:**

You can use `Arrays`, `Structs`, and `Refs` in server-side applications only. You cannot access them in remote clients using the deprecated JDBC over RMI interface.

Programming with Arrays

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

To use an `Array` in WebLogic Server applications:

1. Import the required classes.
2. Get a connection and then create a statement for the connection.
3. Create the `Array` type, a table that uses that type, and create some rows in the table with arrays.
4. Get the `Array` using a result set or a callable statement.
5. Use the standard Java methods (when used as a `java.sql.Array`) or Oracle extension methods (when cast as `java.jdbc.OracleArray`) to work with the data.

The following sections provide more details for these actions:

- [Import Packages to Access Oracle Extensions](#)
- [Establish the Connection](#)
- [Creating an Array in the Database](#)
- [Getting an Array](#)
- [Updating an Array in the Database](#)
- [Using Oracle Array Extension Methods](#)

Import Packages to Access Oracle Extensions

Import the `SQL` and `Oracle` interfaces used in this example.

```
import java.math.BigDecimal;
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import oracle.jdbc.*;
```

Establish the Connection

Establish the database connection using JNDI and `DataSource` objects.

```
// Get a valid DataSource object.
// Here we assume that getDataSource() takes
// care of those details.
javax.sql.DataSource ds = getDataSource(args);
```

```
// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Creating an Array in the Database

You must first create the array type and a table that uses the type. For example:

```
Statement stmt = conn.createStatement();
stmt.execute("CREATE TYPE TEST_SCORES AS VARRAY(10)OF INT");
stmt.execute("CREATE TABLE STUDENTS (STUDENT_ID INT, NAME VARCHAR2(100), SCORES
TEST_SCORES)");
```

The following example creates an array of up to 10 test scores to be associated with a student:

- Create a row with an Array. You can use a Statement or create the Array using `OracleConnection.createOracleArray` for use in a PreparedStatement.

Note:

You cannot use `Connection.createArrayOf` because Oracle does not support anonymous array types

- Insert two rows. The first one uses a SQL statement. The second creates an Array and binds it into a PreparedStatement.

```
stmt.execute("INSERT INTO STUDENTS VALUES 1, 'John Doe', TEST_SCORES(100,99)");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO STUDENTS VALUES
(?, ?, ?)");
pstmt.setInt(1, 2);
pstmt.setString(2, "Jane Doe");
int scores[] = {94, 95};
Array array = ((OracleConnection)conn).createOracleArray("TEST_SCORES", scores);
pstmt.setArray(3, array);
pstmt.execute();
```

Getting an Array

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

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

```
ResultSet rs = null;
rs = stmt.executeQuery("SELECT * FROM STUDENTS");
while (rs.next()) {
    System.out.print("Name="+rs.getString(2)+" : ");
    array = rs.getArray(3);
    BigDecimal scoresBD[] = (BigDecimal[])array.getArray();
    OracleArray oracleArray = (OracleArray)rs.getArray(3);
    scores = oracleArray.getIntArray();
    for (int i = 0; i < scores.length; i++) {
        System.out.print(""+scores[i]+" ");
    }
}
```

```

    }
    System.out.println("");
}

```

Note:

The default return type for an integer is a `BigDecimal`. We can cast the `Array` to an `OracleArray` and use the Oracle extension method `getIntArray()` to get back integer values.

Updating an Array in the Database

To update an `Array` in a database, use the following steps:

1. Create an array in the database, see [Creating an Array in the Database](#).
2. Update the array in the database using the `setArray()` method for a prepared statement or a callable statement. For example:

```

String sqlUpdate = "UPDATE STUDENTS SET SCORES = ? WHERE STUDENT_ID = ?";
int newscores[] = {94, 95, 96};
pstmt = conn.prepareStatement(sqlUpdate);
array = ((OracleConnection)conn).createOracleArray("TEST_SCORES",newscores);
pstmt.setArray(1, array);
pstmt.setInt(2, 1);
pstmt.executeUpdate();

```

Using Oracle Array Extension Methods

To use the Oracle Thin driver extension methods for an `Array`, you must first cast the array as an `oracle.jdbc.OracleArray`. You can then make calls to the Oracle Thin driver extension methods for an `Array` in addition to the standard methods. For example:

```

OracleArray oracleArray = (OracleArray)rs.getArray(3);
String sqltype = oracleArray.getSQLTypeName();

```

Programming with Structs

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

To use a `Struct` in WebLogic Server applications:

1. Import the required classes. (See [Import Packages to Access Oracle Extensions](#).)
2. Get a connection. (See [Establish the Connection](#).)
3. Create the `Struct` object type, a table that uses the object, and rows with `Struct` objects.

4. Cast the object as a `Struct`, either `java.sql.Struct` (to use standard methods) or `oracle.jdbc.OracleStruct` (to use standard and Oracle extension methods).
5. Use the standard or Oracle Thin driver extension methods to work with the data.

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

- [Creating Objects in the Database](#)
- [Getting Struct Attributes](#)
- [Using OracleStruct Extension Methods](#)
- [Using a Struct to Update Objects in the Database](#)

Creating Objects in the Database

A `Struct` is typically used to materialize database objects in your Java application in place of custom Java classes that map to the database objects. You must first create the type and table that uses the type. For example (this snippet is poorly designed and used for demonstration purposes only):

```
conn = ds.getConnection();
Statement stmt = conn.createStatement();
stmt.execute("CREATE TYPE EMP_STRUCT AS OBJECT (DEPT INT, NAME VARCHAR2(100))");
stmt.execute("CREATE TABLE EMP (ID INT, EMPLOYEE EMP_STRUCT)");
```

To create a row with a `Struct` object, you can use a `SQL Statement` or create the `Struct` using `Connection.createStruct` and use it in a `PreparedStatement`.

Insert two rows. The first one row uses a `SQL statement`. The second creates a `Struct` and binds it into a `PreparedStatement`.

```
stmt.execute("INSERT INTO EMP VALUES (1001, EMP_STRUCT(10, 'John Doe'))");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO EMP VALUES (?,?)");
Object attrs[] = { new Integer(20), "Jane Doe"};
Struct struct = conn.createStruct("EMP_STRUCT", attrs);
pstmt.setInt(1,1002);
pstmt.setObject(2,struct);
pstmt.execute();
```

Note:

When creating a `SQL structure` using `Connection.createStruct()`, it is necessary to unwrap all data types (`Clob`, `Blob`, `Struct`, `Ref`, `Array`, `NClob`, and `SQLXML`). Once the structure is created, there is no way to re-wrap them before returning the structure to the application. The structure returned to the application has unwrapped values for the data types.

Getting Struct Attributes

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

You can create a result set, get a `Struct` from the result set, and then use the `getAttributes()` method. The method returns an array of ordered attributes. You can assign the attributes from the `Struct` (object in the database) to an object in the

application, including Java language types. You can then manipulate the attributes individually. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs = stmt.executeQuery("SELECT * FROM EMP WHERE ID = 1002");
//The second column uses an object data type.
if (rs.next()) {
    struct = (Struct)rs.getObject(2);
    attrs = struct.getAttributes();
    String name = attrs[1];
}
```

In the preceding example, the second column in the `emp` table uses an object data type. The example shows how to assign the results from the `getObject` method to a Java object that contains an array of values, and then use individual values in the array as necessary. Note that the type of the first integer attribute is actually a `java.math.BigDecimal`.

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

```
java.util.Hashtable map = new java.util.Hashtable();
map.put("INT", Class.forName("java.lang.Integer"));
map.put("VARCHAR2", Class.forName("java.lang.String"));
Object[] attrs = struct.getAttributes(map);
String name = (String)attrs[1];
```

In this example, the value is returned as an `Integer` instead of a `BigDecimal`.

Using OracleStruct Extension Methods

To use the Oracle Thin driver extension methods for a `Struct`, you must cast the `java.sql.Struct` (or the original `getObject` result) as a `oracle.jdbc.OracleStruct`. When you cast a `Struct` as an `OracleStruct`, you can use both the standard and extension methods. For example:

```
OracleStruct oracleStruct =
    (OracleStruct)rs.getObject(2);
String n = oracleStruct.getSQLTypeName(); // Standard
oracle.jdbc.OracleTypeMetaData otmd =
    oracleStruct.getOracleMetaData(); // Extension
```

Using a Struct to Update Objects in the Database

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

```
pstmt = conn.prepareStatement("UPDATE EMP SET EMPLOYEE = ? WHERE ID =?");
attrs[0] = new Integer(30);
struct = conn.createStruct("EMP_STRUCT", attrs);
pstmt.setObject (1, struct);
pstmt.setInt (2, 1002);
pstmt.executeUpdate();
```

Programming with Refs

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

To use a `Ref` in WebLogic Server applications, use the following steps:

1. Import the required classes. (See [Import Packages to Access Oracle Extensions.](#))
2. Get a database connection. (See [Establish the Connection.](#))
3. Create a `Ref` using a SQL Statement.
4. Get the `Ref` using a result set or a callable statement.
5. Use the extended Oracle methods by casting to `OracleRef`.
6. Update a `Ref` in the database.

The following sections describe steps 3 through 6 in greater detail:

- [Creating a Ref in the Database](#)
- [Getting a Ref](#)
- [Using WebLogic OracleRef Extension Methods](#)
- [Updating Ref Values](#)

Creating a Ref in the Database

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

```
conn = ds.getConnection();
stmt = conn.createStatement();
stmt.execute("CREATE TYPE OB AS OBJECT (OB1 INT, OB2 INT)");
stmt.execute("CREATE TABLE T1 OF OB");
stmt.execute("INSERT INTO T1 VALUES (5, 5)");
stmt.execute("CREATE TABLE T2 (COL REF OB)");
stmt.execute("INSERT INTO T2 SELECT REF(P) FROM T1 P WHERE P.OB1=5");
```

The preceding example creates an object type (`OB`), a table (`T1`) of that object type, a table (`T2`) with a `Ref` column that can point to instances of `OB` objects, and inserts a `Ref` into the `Ref` column. The `Ref` points to a row in `T1` where the value in the first column is 5.

Getting a Ref

To get a `Ref` in an application, you can use a query to create a result set and then use the `getRef` method to get the `Ref` from the result set. For example:

```
rs = stmt.executeQuery("SELECT REF (S) FROM T1 S WHERE S.OB1=5");
rs.next();
Ref ref = rs.getRef(1);
String name = ref.getBaseTypeName();
```

The `WHERE` clause in the preceding example uses dot notation to specify the attribute in the referenced object. After you get the `Ref`, you can use the Java API method `getBaseTypeName`.

Using WebLogic OracleRef Extension Methods

In order to use the Oracle Thin driver extension methods for `Refs`, you must cast the `Ref` as an `OracleRef`. For example:

```
OracleTypeMetaData mdata = ((OracleRef)ref). getOracleMetaData();
```

Updating Ref Values

To update a `Ref`, you change the location to which the `Ref` points with a `PreparedStatement` or a `CallableStatement`.

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

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

For example:

```
//Get the Ref
rs = stmt.executeQuery("SELECT REF (S) FROM T1 S WHERE S.OB1=5");
rs.next();
ref = rs.getRef(1);
//Create and execute the prepared statement.
String sqlUpdate = "UPDATE T2 S2 SET COL = ? WHERE S2.COL.OB1 = 20";
pstmt = conn.prepareStatement(sqlUpdate);
pstmt.setRef(1, ref);
pstmt.executeUpdate();
```

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

```
rs = stmt.executeQuery("SELECT REF (S) FROM T1 S where S.OB1=5");
rs.next();
ref = rs.getRef(1);
// Prepare the stored procedure
String sql = "{call SP1 (?,?)}";
CallableStatement cstmt = conn.prepareCall(sql);
// Set IN and register OUT params
cstmt.setRef(1, ref);
cstmt.registerOutParameter(2, Types.STRUCT, "OB");
// Execute
cstmt.execute();
```


Programming with Large Objects

This section contains information, including sample code, on how to work with Blob and Clob objects. For additional information, refer to *Working with LOBs in Database SecureFiles and Large Objects Developer's Guide*.

- [Creating Blobs in the Database](#)
- [Updating Blobs in the Database](#)
- [Using OracleBlob Extension Methods](#)
- [Programming with Clob Values](#)
- [Transaction Boundaries Using LOBs](#)
- [Recovering LOB Space](#)

Creating Blobs in the Database

The following code presumes the `Connection` is already established. It creates a table with a Blob as the second column.

```
ResultSet rs = null;
Statement stmt = null;
java.sql.Blob blob = null;
java.io.InputStream is = null;
stmt = conn.createStatement();
stmt.execute("CREATE TABLE TESTBLOB (ID INT, COL2 BLOB )");
```

The following code inserts a Blob value using a string converted to a byte array as the data.

```
String insertsql2 = "INSERT INTO TESTBLOB VALUES (?,?)";
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO TESTBLOB VALUES (?,?)");
pstmt.setInt(1, 1);
pstmt.setBytes(2, "initialvalue".getBytes());
pstmt.executeUpdate();
```

Updating Blobs in the Database

The following code updates the Blob value.

```
rs = stmt.executeQuery("SELECT COL2 FROM TESTBLOB WHERE ID = 1 FOR UPDATE");
rs.next();
Blob blob = rs.getBlob(1);
blob.setBytes(1, "newdata".getBytes());
```

Note that you need the `FOR UPDATE` to be able to update the Blob value.

Using OracleBlob Extension Methods

The following code casts the Blob to an `OracleBlob` so that you can use an extension method.

```
rs = stmt.executeQuery("SELECT COL1, COL2 FROM TESTBLOB");
rs.next();
Blob blob = rs.getBlob(2);
```

```
is = blob.getBinaryStream();           // Standard
is.close();
is = ((OracleBlob)blob).getBinaryStream(0); // Extended
```

Once you cast to the `OracleBlob` interface, you can access the Oracle supported methods in addition to the standard methods. `LOB#freeTemporary` should be replaced with `OracleBlob#free`.

Programming with Clob Values

Using `Clob` values is similar to using `Blob` values except that the data is a string instead of a binary array (use `setString` instead of `setBytes`, `getClob` instead of `getBlob`, and `getCharacterStream` instead of `getBinaryStream`).

If you use a prepared statement to update a `Clob` and the new value is shorter than the previous value, the `Clob` retains the characters that were not specifically replaced during the update. For example, if the current value of a `Clob` is `abcdefghijkl` and you update the `Clob` using a prepared statement with `zxyw`, the value in the `Clob` is updated to `zxywefghij`. To correct values updated with a prepared statement, you should use the `dbms_lob.trim` procedure to remove the excess characters left after the update. See `DBMS_LOB` in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `dbms_lob.trim` procedure. `LOB#freeTemporary` must be replaced with `OracleClob#free`.

Transaction Boundaries Using LOBs

When using LOBs, you must take transaction boundaries into account; for example, direct all read/writes to a particular LOB within a transaction.

Recovering LOB Space

To free up space used by a LOB, it's necessary to call `lob.close()`. This is not automatically done when a `ResultSet`, `Statement`, or `Connection` is closed. For Oracle data bases only, it is also necessary to execute `alter session set events '60025 trace name context forever';` on the session so that other sessions can use the freed memory.

Programming with Opaque Objects

This topic describes the use case of working with Opaque Objects.

The new Oracle type interfaces have only methods that are considered significant or not available with standard JDBC API's. Here the `oracle.sql.OPAQUE` has been replaced with `oracle.jdbc.OracleOpaque`. The new interface only has a method to get the value as an Object and two meta information methods to get meta data and type name. Unlike the other Oracle type interfaces (`oracle.jdbc.OracleStruct` extends `java.sql.Struct` and `oracle.jdbc.OracleArray` extends `java.sql.Array`), `oracle.jdbc.OracleOpaque` does not extend a JDBC interface.

Since `XMLType` doesn't work with the replay datasource and the `oracle.xdb` package uses `XMLType` extensively, this package is no longer usable for Application Continuity replay.

There is one related very common use case that needs to be changed to work with Application Continuity (AC). Early uses of SQLXML made use of the following XDB API.

```
SQLXML sqlXml = oracle.xdb.XMLType.createXML(
((oracle.jdbc.OracleResultSet)resultSet).getOPAQUE("issue"));
```

oracle.xdb.XMLType extends oracle.sql.OPAQUE and its use will disable AC replay. This must be replaced with the standard JDBC API

```
SQLXML sqlXml = resultSet.getSQLXML("issue");
```

The JDeveloper JPublisher feature has been deprecated and removed starting in Release 12.2.1. Code generated by this feature includes concrete classes, requiring the re-write of the code as described above. Here are several additional hints on doing that re-write.

MutableArray#toDatum should be replaced with OracleDataMutableArray.toJDBCObject.

MutableStruct#toDatum should be replaced with OracleDataMutableStruct.toJDBCObject. The following are the additional classes that have new interfaces. They do not have corresponding WLS interfaces and they do not map to JDBC types.

-	-
oracle.sql.ORAData	oracle.jdbc.OracleData
oracle.sql.ORADataFactory	oracle.jdbc.OracleDataFactory
oracle.sql.OPAQUE	oracle.jdbc.OracleOpaque
oracle.sql.NCLOB	oracle.jdbc.OracleNClob
oracle.sql.BFILE	oracle.jdbc.OracleBfile
oracle.sql.Datum	java.lang.Object and then use instanceOf for other interface types
oracle.jpub.runtime.MutableStruct	oracle.jpub.runtime.OracleDataMutabl eStruct
oracle.jpub.runtime.MutableArray	oracle.jpub.runtime.OracleDataMutabl eArray

Using Batching with the Oracle Thin Driver

In some situations, the Oracle Thin driver may not send updates to the DBMS if a batch size has not been reached and waits until the statement is closed. When a Prepared Statement is closed, WebLogic Server returns the statement to a standard JDBC state rather than closing it. It is then put back into the pool for the connection so it can be re-delivered the next time it is needed.

To make sure all your updates are delivered, you need to call `OraclePreparedStatement.sendBatch()` explicitly after the last use of the statement, before closing it or closing the connection.

Using the Java Security Manager with the Oracle Thin Driver

Learn how to use Java Security Manager with Oracle Thin Driver to create a security policy for an application.

When using the Oracle Thin Driver with the Java Security Manager enabled, it is necessary to update privileges in your `java.policy` file.

1. Download the `Demo` jar file for the [Oracle JDBC driver](#) from the Oracle Technology Network.
2. Review the `ojdbc.policy` file, it specifies the permissions required for the driver.
3. Add these privileges to the policy file used to run the server. For example, `java.util.PropertyPermission "oracle.jdbc.*", "read";` is required for the `ojdbc.jar` file.

8

Getting a Physical Connection from a Data Source

To directly access a physical connection from a data source, standard practice is to cast a connection to the generic JDBC connection (a wrapped physical connection) provided by WebLogic Server. Oracle strongly discourages directly accessing a physical JDBC connection except for when it is absolutely required.

The standard practice of casting a connection to the generic JDBC connection allows the server instance to manage the connection for the connection pool, enable connection pool features, and maintain the quality of connections provided to applications. Occasionally, a DBMS provides extra non-standard JDBC-related classes that require direct access of the physical connection (the actual vendor JDBC connection). To directly access a physical connection in a connection pool, you must cast the connection using `getVendorConnection`.

Note:

Oracle also provides another mechanism to access a physical connection `getVendorConnectionSafe`. This mechanism also returns the underlying physical connection (the vendor connection) from a pooled database connection (a logical connection). However, when the connection is closed, it is returned to the pool, independent of the setting of `Remove Infected Connections Enabled`. See `getVendorConnectionSafe`.

This chapter includes the following sections:

- [Opening a Connection](#)
- [Closing a Connection](#)
- [Limitations for Using a Physical Connection](#)

Note:

Oracle strongly discourages directly accessing a physical JDBC connection except for when it is absolutely required.

Opening a Connection

To get a physical connection, you first need to get a connection from a connection pool and then implicitly pass the physical connection or cast the connection.

After obtaining a connection from a connection pool, do one of the following:

- Implicitly pass the physical connection (using `getVendorConnection`) within a method that requires the physical connection.
- Cast the connection as a `WLConnection` and call `getVendorConnection`.

Always limit direct access of physical database connections to vendor-specific calls. For all other situations, use the generic JDBC connection provided by WebLogic Server. Sample code to open a connection for vendor-specific calls is provided below.

Example 8-1 Code Sample to Open a Connection for Vendor-specific Calls

```
//Import this additional class and any vendor packages
//you may need.
import weblogic.jdbc.extensions.WLConnection
.
.
.
myJdbcMethod()
{
    // Connections from a connection pool should always be
    // method-level variables, never class or instance methods.
    Connection conn = null;
    try {
        ctx = new InitialContext(ht);
        // Look up the data source on the JNDI tree and request
        // a connection.
        javax.sql.DataSource ds
            = (javax.sql.DataSource) ctx.lookup ("myDataSource");
        // Always get a pooled connection in a try block where it is
        // used completely and is closed if necessary in the finally
        // block.
        conn = ds.getConnection();
        // You can now cast the conn object to a WLConnection
        // interface and then get the underlying physical connection.
        java.sql.Connection vendorConn =
            ((WLConnection)conn).getVendorConnection();
        // do not close vendorConn
        // You could also cast the vendorConn object to a vendor
        // interface, such as:
        // oracle.jdbc.OracleConnection vendorConn = (OracleConnection)
        // ((WLConnection)conn).getVendorConnection()
        // If you have a vendor-specific method that requires the
        // physical connection, it is best not to obtain or retain
        // the physical connection, but simply pass it implicitly
        // where needed, eg: //
        vendor.special.methodNeedingConnection(((WLConnection)conn).getVendorConnection(
    ));
    }
```

Closing a Connection

Once you have completed the JDBC work, you should close the logical connection in order to return the connection to the pool.

When you are done with the physical connection:

- Close any objects you have obtained from the connection.
- Do not close the physical connection. Set the physical connection to null.
- You determine how a connection closes by setting the value of the `Remove Infected Connections Enabled` property in the WebLogic Server Administration

Console. See the [JDBC Data Source: Configuration: Connection Pool](#) page in the *Oracle WebLogic Server Administration Console Online Help* or see [JDBCConnectionPoolParamsBean](#) in the *MBean Reference for Oracle WebLogic Server* for more details about these options.

 **Note:**

The Remove Infected Connections Enabled property applies only to applications that explicitly call `getVendorConnection`.

Example 8-2 Sample Code to Close a Connection for Vendor-specific Calls

```
// As soon as you are finished with vendor-specific calls,
// nullify the reference to the connection.
// Do not keep it or close it.
// Never use the vendor connection for generic JDBC.
// Use the logical (pooled) connection for standard JDBC.
vendorConn = null;
... do all the JDBC needed for the whole method...
// close the logical (pooled) connection to return it to
// the connection pool, and nullify the reference.
conn.close();
conn = null;
}
catch (Exception e)
{
    // Handle the exception.
}
finally
{
    // For safety, check whether the logical (pooled) connection
    // was closed.
    // Always close the logical (pooled) connection as the
    // first step in the finally block.
    if (conn != null) try {conn.close();} catch (Exception ignore){}
}
}
```

Remove Infected Connections Enabled is True

When `Remove infected Connections Enabled=true` (default value) and you close the logical connection, the server instance discards the underlying physical connection and creates a new connection to replace it. This action ensures that the pool can guarantee to the next user that they are the sole user of the physical connection. This configuration provides a simple and safe way to close a connection. However, there is a performance loss because:

- The physical connection is replaced with a new database connection in the connection pool, which uses resources on both the application server and the database server.
- The statement cache for the original connection is closed and a new cache is opened for the new connection. Therefore, the performance gains from using the statement cache are lost.

Remove Infected Connections Enabled is False

Use `Remove infected Connections Enabled=false` only if you are sure that the exposed physical connection will never be retained or reused after the logical connection is closed.

When `Remove infected Connections Enabled=false` and you close the logical connection, the server instance simply returns the physical connection to the connection pool for reuse. Although this configuration minimizes performance losses, the server instance does not guarantee the quality of the connection or to effectively manage the connection after the logical connection is closed. You must make sure that the connection is suitable for reuse by other applications before it is returned to the connection pool.

Limitations for Using a Physical Connection

Learn about the limitations of using a physical connection instead of a logical connection from a connection pool.

Oracle strongly discourages using a physical connection instead of a logical connection from a connection pool. However, if you must use a physical connection, for example, to create a `STRUCT`, consider the following costs and limitations:

- The physical connection can only be used in server-side code.
- When you use a physical connection, you lose all of the connection management benefits that WebLogic Server offer, such as error handling and statement caching.
- You should use the physical connection only for the vendor-specific methods or classes that require it. Do not use the physical connection for generic JDBC, such as creating statements or transactional calls.

9

Using RowSets with WebLogic Server

Learn about the characteristics and usage of WebLogic RowSets such as types of RowSets, RowSet extensions, usage of RowSet in programming and so on.

- [Deprecation of `weblogic.jdbc.rowsets`](#)
- [About RowSets](#)
- [Types of RowSets](#)
- [Programming with RowSets](#)
- [CachedRowSets](#)
- [RowSet MetaData Settings for Database Updates](#)
- [WebLogic RowSet Extensions for Working with MetaData](#)
- [RowSets and Transactions](#)
- [FilteredRowSets](#)
- [WebRowSets](#)
- [JoinRowSets](#)
- [JDBCRowSets](#)
- [Handling `SyncProviderExceptions` with a `SyncResolver`](#)
- [WLCachedRowSets](#)
- [SharedRowSets](#)
- [SortedRowSets](#)
- [SQLPredicate, a SQL-Style RowSet Filter](#)
- [Optimistic Concurrency Policies](#)
- [Performance Options](#)

Deprecation of `weblogic.jdbc.rowsets`

The `weblogic.jdbc.rowset` interfaces and classes are deprecated in WebLogic Server 12.1.2.

Use the reference implementation, `com.sun.rowset`, in the J2SE JRE/SDK. See [weblogic.jdbc.rowset](#)

About RowSets

A RowSet is an extension of a Java ResultSet. Like a ResultSet, a rowset is a Java object that holds tabular data. However, a rowset adds significant flexibility to ResultSet features and reduces or eliminates some ResultSet limitations.

WebLogic Server includes an implementation of Java RowSets according to the specifications indicated in JSR-114. See (<http://www.oracle.com/technetwork/java/javase/jdbc/index.html>) for details about the specification. The WebLogic rowset implementation also includes extensions to the RowSets specification. These extensions make RowSets more useful in your applications.

Types of RowSets

The WebLogic Server implementation of rowsets includes standard RowSet types and WebLogic RowSet extensions.

Standard RowSet Types:

- [CachedRowSets](#)
- [FilteredRowSets](#)
- [WebRowSets](#)
- [JoinRowSets](#)
- [JDBCRowSets](#)

WebLogic RowSet Extensions:

- [WLCachedRowSets](#)
- [SharedRowSets](#)
- [SortedRowSets](#)
- [SQLPredicate, a SQL-Style RowSet Filter](#)

Programming with RowSets

The WebLogic RowSet implementation includes a life cycle framework that prevents a rowset object from getting into an unhealthy condition.

The WebLogic Server rowset implementation was designed with the expectation that you would work with a rowset in the following manner:

1. Create and configure the rowset — define the query, database connection, and other properties.
2. Populate the rowset with data — specify query parameters and execute the query.
3. Optionally, work with rowset metadata.
4. Optionally set the filter or sorter for the rowset.
5. Manipulate the data in the rowset — insert, update, and delete.
6. Synchronize data changes from the rowset to the database.

After synchronizing changes, the process can repeat starting with step 2 or 3, depending on the way your application is designed. See [Reusing a WebLogic RowSet After Completing a Transaction](#).

Internally, WebLogic Server sets a life cycle stage for the RowSet as the RowSet moves through the previously described process. To reduce the risk of data loss, WebLogic Server limits the operations you can do on the rowset depending on the life cycle stage of the rowset. For example, when the RowSet is in the *Updating* stage,

you can only call `update XXX ()` methods, such as `updateString()` and `updateInt()`, on the `RowSet` until you call `updateRow()` to complete the update phase.

Some important notes:

- If you have pending changes, you cannot re-populate, filter, or sort the `RowSet`. WebLogic Server prevents these operations on the `RowSet` when the `RowSet` data has changed but the changes have not been synchronized with the database to prevent the accidental loss of data changes.
- There is no implicit movement of the cursor! You must explicitly move the cursor from row to row.
- `RowSet` life cycle stage is an internal process. There are no public APIs to access it. You cannot set the life cycle stage. When you call `acceptChanges()` or `restoreOriginal()`, WebLogic Server resets the life cycle stage of the `RowSet` so you can begin again.

 **Note:**

When using a rowset in a client-side application, the *exact same* JDBC driver classes must be in the `CLASSPATH` on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

See the comments in [Example 9-1](#) for an illustration of the life cycle stages for a rowset from when it is created to when data changes are synchronized with the database.

CachedRowSets

Learn about using standard `CachedRowSets` with WebLogic Server.

- [Characteristics](#)
- [Special Programming Considerations and Limitations for CachedRowSets](#)
- [Code Example](#)
- [Importing Classes and Interfaces for a CachedRowSet](#)
- [Creating a CachedRowSet](#)
- [Setting CachedRowSet Properties](#)
- [Database Connection Options](#)
- [Populating a CachedRowSet](#)
- [Setting CachedRowSet MetaData](#)
- [Working with Data in a CachedRowSet](#)
- [Synchronizing RowSet Changes with the Database](#)

Also see [WLCachedRowSets](#) for information about using WebLogic extensions to the standard `CachedRowSet` object.

Characteristics

A `CachedRowSet` is a disconnected `ResultSet` object. Data in a `CachedRowSet` is stored in memory. `CachedRowSets` from the WebLogic Server implementation have the following characteristics:

- Can be used to insert, update, or delete data.
- Are serializable, so they can be passed to various application components, including wireless devices.
- Include transaction handling to enable rowset reuse. See [Reusing a WebLogic RowSet After Completing a Transaction](#).
- Use an optimistic concurrency control for synchronizing data changes in the rowset with the database.
- Use a `SyncResolver` object from a `SyncProvider` exception to resolve conflicts between data changes in the rowset and the database. See [Handling SyncProviderExceptions with a SyncResolver](#).

Special Programming Considerations and Limitations for CachedRowSets

When designing your application, consider the following information:

- Entire RowSet Query Results Stored in Memory
- Data Contention

Entire RowSet Query Results Stored in Memory

Because a `CachedRowSet` does not hold a connection to the database, it must hold the entire query results in memory. If the query result is very large, you may see performance degradation or out-of-memory errors. For large data sets, a `ResultSet` may be more appropriate because it keeps a connection to the database, so it can hold partial query results in memory and return to the database for additional rows as needed.

Data Contention

`CachedRowSets` are most suitable for use with data that is not likely to be updated by another process between when the rowset is populated and when data changes in the rowset are synchronized with the database. Database changes during that period will cause data contention. See [Handling SyncProviderExceptions with a SyncResolver](#) for more information about detecting and handling data contention.

Code Example

[Example 9-1](#) shows the basic workflow of a `CachedRowSet`. It includes comments that describe each major operation and its corresponding rowset life cycle stage. Following the code example is a more detailed explanation of each of the major sections of the example.

Example 9-1 Cached RowSet Code Example

```
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetFactory;
public class CachedRowSetDemo {
public static void main (String[] args) {
//DESIGNING lifecycle stage - Create the rowset and set properties
try {
//Create a RowSetFactory instance and from the factory,
//create a FilteredRowSet.
RowSetFactory rsfact =
RowSetProvider.newFactory("weblogic.jdbc.rowset.JdbcRowSetFactory",null);
CachedRowSet rs = rsfact.createCachedRowSet();
//Set database access through a DataSource.
rs.setDataSourceName(examples-datasource-demoPool);
//See Database Connection Options for more options.
//Set query command
rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
PHONE, EMAIL FROM PHYSICIAN WHERE ID>?");
//CONFIGURE QUERY lifecycle operation
rs.setInt(1, 0);
//POPULATING lifecycle stage - Execute the command to populate the rowset
rs.execute();
}
//CONFIGURING METADATA - Populate first, then set MetaData,
//including KeyColumns
rs.setKeyColumns(new int[] { 1 });
while (rs.next ()) //NAVIGATING lifecycle stage
{
System.out.println ("ID: " +rs.getInt (1));
System.out.println ("FIRST_NAME: " +rs.getString (2));
System.out.println ("MIDDLE_NAME: " +rs.getString (3));
System.out.println ("LAST_NAME: " +rs.getString (4));
System.out.println ("PHONE: " +rs.getString (5));
System.out.println ("EMAIL: " +rs.getString (6));
}
}
//Working with data
//Delete rows in the rowset
try {
//MANIPULATING lifecycle stage - navigate to a row
//(manually moving the cursor)
rs.last();
rs.deleteRow();
//Note that the database is not updated yet.
}
//Update a row in the rowset
try {
//MANIPULATING lifecycle stage - navigate to a row
//(manually moving the cursor)
rs.first();
//UPDATING lifecycle stage - call an update() method
rs.updateString(4, "Francis");
//MANIPULATING lifecycle stage - finish update
rs.updateRow();
//Note that the database is not updated yet.
}
//INSERTING lifecycle stage - Insert rows in the rowset
try {
rs.moveToInsertRow();
rs.updateInt(1, 104);
}
```

```

rs.updateString("FIRST_NAME", "Yuri");
rs.updateString("MIDDLE_NAME", "M");
rs.updateString("LAST_NAME", "Zhivago");
rs.updateString("PHONE", "1234567812");
rs.updateString("EMAIL", "Yuri@poet.com");
rs.insertRow(); // "Finish Update" action;
// MANIPULATING lifecycle stage - navigate to a row
rs.moveToCurrentRow();
// Note that the database is not updated yet.
}
// Send all changes (delete, update, and insert) to the database.
// DESIGNING or POPULATING lifecycle stage - after synchronizing changes
// with the database, lifecycle stage depends on other environment settings.
// See Reusing a WebLogic RowSet After Completing a Transaction.
try {
    rs.acceptChanges();
    rs.close();
}
}

```

Importing Classes and Interfaces for a CachedRowSet

For standard RowSets, you must import the following classes:

```

javax.sql.rowset.CachedRowSet;
javax.sql.rowset.RowSetFactory;

```

Creating a CachedRowSet

Rowsets are created from a factory interface. To create a rowset with WebLogic Server, follow these main steps:

1. Create a RowSetFactory instance, which serves as a factory to create rowset objects for use in your application.

```

RowSetFactory rsfact =
RowSetProvider.newFactory("weblogic.jdbc.rowset.JdbcRowSetFactory", null);

```

2. Create a javax.sql.rowset.CachedRowSet object. For example:

```

CachedRowSet rs = rsfact.createCachedRowSet();

```

Setting CachedRowSet Properties

There are numerous rowset properties, such as concurrency type, data source name, transaction isolation level, and so forth, that you can set to determine the behavior of the rowset. You are required to set only those properties that are needed for your particular use of the rowset. For information about available properties, see the Javadoc for the javax.sql.rowset.BaseRowSet class at <http://docs.oracle.com/javase/1.5.0/docs/api/javax/sql/rowset/BaseRowSet.html>.

Database Connection Options

In most applications, you populate a rowset with data from a database. You can set rowset database connectivity in any of the following ways:

- Automatically with a data source—You can use the setDataSourceName() method to specify the JNDI name of a JDBC data source. When you call execute() and

`acceptChanges()`, the rowset gets a database connection from the data source, uses it, and returns it to the pool of connections in the data source. This is a preferred method.

```
rs.setDataSourceName(examples-datasource-demoPool);
```

- **Manually get a database connection**—In your application, you can get a database connection before the rowset needs it, and then pass the connection object as a parameter in the `execute()` and `acceptChanges()` methods. You must also close the connection as necessary.

```
//Lookup DataSource and get a connection
ctx = new InitialContext(ht);
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup ("myDS");
conn = ds.getConnection();
```

```
//Pass the connection to the rowset
rs.execute(conn);
```

For more information about JDBC data sources, see [Getting a Database Connection from a DataSource Object](#).

- **Load the JDBC driver for a direct connection**—When you load the JDBC driver and set the appropriate properties, the rowset creates a database connection when you call `execute()` and `acceptChanges()`. The rowset closes the connection immediately after it uses it. The rowset does not keep the connection between the `execute()` and `acceptChanges()` method calls.

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
rs.setUrl("jdbc:derby://localhost:1527/demo");
rs.setUsername("examples");
rs.setPassword("examples");
rs.execute();
```

Populating a CachedRowSet

Populating a rowset is the act of filling the rowset with rows of data. The source of the data is most commonly a relational database. To populate a rowset with data from a database, you can use either of the following methods:

- **Set an SQL command with the `setCommand()` method, then execute the command with the `execute()` method:**

```
rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
              PHONE, EMAIL FROM PHYSICIAN");
rs.execute();
```

- **From an existing result set using the `populate()` method:**

```
rs.populate(resultSet);
```

 **Note:**

If using a result set that is `ResultSet.TYPE_FORWARD_ONLY`, a `SQLException` will be thrown if you attempt to populate a row set with the following conditions:

- If you call `CachedRowset.populate(ResultSet rs)` when the result set cursor is at a position beyond row 1.
- If you call `CachedRowset.populate(ResultSet rs, int newPosition)` when `newPosition` is less than the current result set cursor position.

Setting CachedRowSet MetaData

In some cases, you may need to set metadata for the rowset in order to synchronize data changes in the rowset with data in the database. See [RowSet MetaData Settings for Database Updates](#).

Working with Data in a CachedRowSet

After you populate the cached rowset with rows of data, you can work with the cached data in much the same way as you would work with data in a result set, except that before your changes are made in the database, you must explicitly call `acceptChanges()`.

 **Note:**

Delimiter identifiers may not be used for column or table names in rowsets. Delimiter identifiers are identifiers that need to be enclosed in double quotation marks when appearing in a SQL statement. They include identifiers that are SQL reserved words (e.g., `USER`, `DATE`, etc.) and names that are not identifiers. A valid identifier must start with a letter and contain only letters, numbers, and underscores.

Getting Data from a Row in a RowSet

To get data from a rowset, you use the `getXXX` methods just as you would with a result set. For example:

```
while (rs.next ())
{
    int id = rs.getInt (1);
    String fname = rs.getString ("FIRST_NAME");
    String mname = rs.getString ("MIDDLE_NAME");
    String lname = rs.getString ("LAST_NAME");
}
```

Updating a Row in a RowSet

Data updates typically follow this course of events:

1. Navigate to the row or to an insert row.
2. Change the row with `updateXXX` methods.
3. Complete the operation with `updateRow()` or `insertRow()`.

Note that completing the operation does not synchronize your changes with the database. Changes are made to the rowset only. You must explicitly synchronize your changes by calling `acceptChanges()`. For details, see [Synchronizing RowSet Changes with the Database](#) later in this section.

When working with a rowset, WebLogic Server internally sets the life cycle stage of the rowset after each operation on the rowset, and then limits further operations you can perform on the rowset based on its current life cycle stage. After you begin modifying a row with update methods, you must complete the operation with `updateRow()` or `insertRow()` before you can work with data in any other rows, including moving the cursor to another row. See [Programming with RowSets](#) for a complete discussion of rowset life cycle stages and operations allowed for each stage.

To update a row, you move the cursor to the row you want to update, call `updateXXX` methods on individual columns within the row, then call `updateRow()` to complete the operation. For example:

```
rs.first();
rs.updateString(4, "Francis");
rs.updateRow();
```

 **Note:**

If you are updating same-named columns from more than one table, you must use the column index number to refer to the column in the update statement.

Inserting a Row in a RowSet

To insert a row, you move the cursor to a new insert row, update the column values within the row, then call `insertRow()` to add the row to the rowset. For example:

```
rs.moveToInsertRow();
rs.updateInt(1, 104);
rs.updateString("FIRST_NAME", "Yuri");
rs.updateString("MIDDLE_NAME", "M");
rs.updateString("LAST_NAME", "Zhivago");
rs.updateString("PHONE", "1234567812");
rs.updateString("EMAIL", "Yuri@poet.com");
rs.insertRow();
rs.moveToCurrentRow();
```

Note that you must explicitly move the cursor after inserting a row. There is no implicit movement of the cursor.

Deleting a Row in a RowSet

To delete a row in the rowset, you move the cursor to the row and call `deleteRow()`. For example:

```
rs.last();  
rs.deleteRow();
```

Synchronizing RowSet Changes with the Database

After you make changes to individual rows in a rowset, you call `acceptChanges()` to propagate those changes to the database. For example:

```
rs.acceptChanges();
```

When you call `acceptChanges()`, the rowset connects to the database using the database connection information already used by the rowset (see [Database Connection Options](#)) or using a connection object passed with the `acceptChanges(connection)` method. You can call `acceptChanges()` after making changes to one row or several rows. Calling `acceptChanges()` after making all changes to the rowset is more efficient because the rowset connects to the database only once.

When using rowsets with WebLogic Server, WebLogic Server internally uses a `weblogic.jdbc.rowset.WLSyncProvider` object to read from and write to the database. The `WLSyncProvider` uses an optimistic concurrency algorithm for making changes to the database, which means that the design assumes data in the database will not be changed by another process during the time between when a rowset is populated to when rowset data changes are propagated to the database. Before writing changes to the database, the `WLSyncProvider` compares the data in the database against the original values in the rowset (values read into the rowset when the rowset was created or at the last synchronization). If any values in the database have changed, WebLogic Server throws a `javax.sql.rowset.spi.SyncProviderException` and does not write any changes to the database. You can catch the exception in your application and determine how to proceed. See [Handling SyncProviderExceptions with a SyncResolver](#).

The `WLCachedRowSet` interface, an extension to the standard `CachedRowSet` interface, provides options for selecting an optimistic concurrency policy. See [Optimistic Concurrency Policies](#).

After propagating changes to the database, WebLogic Server changes the life cycle stage of the rowset to *Designing* or *Populating*, depending on your application environment. In the *Designing* stage, you must repopulate the rowset before you can use it again; in the *Populating* stage, you can use the rowset with its current data. See [Reusing a WebLogic RowSet After Completing a Transaction](#) for more details.

If you do not plan to use the rowset again, you should close it with the `close()` method. For example:

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

RowSet MetaData Settings for Database Updates

When populating a rowset with an SQL query, the WebLogic rowset implementation uses the `ResultSetMetaData` interface to automatically learn the table and column names of the data in the rowset. In many cases, this is enough information for the rowset to generate the required SQL to write changes back to the database. However, some JDBC drivers do not include table and column metadata for the rows returned by the query.

When you attempt to synchronize data changes in the rowset with the database, you will see the following error:

```
java.sql.SQLException: Unable to determine the table name for column:
column_name. Please ensure that you've called WLRowSetMetaData.setTableName to
set a table name for this column.
```

Without the table name, you can use the rowset for read-only operations only. The rowset cannot issue updates unless the table name is specified programmatically. You may also need to set the primary key columns with the `setKeyColumns()` method. For example:

```
rs.setTableName(PHYSICIAN);
rs.setKeyColumns(new int[] { 1 });
```

See the documentation for the `javax.sql.rowset.CachedRowSet` interface for more details.

WebLogic RowSet Extensions for Working with MetaData

Learn about WebLogic rowset extensions that you can use to obtain or set the appropriate metadata for a rowset.

- [executeAndGuessTableName](#) and [executeAndGuessTableNameAndPrimaryKeys](#)
- [Setting Table and Primary Key Information Using the MetaData Interface](#)
- [Setting the Write Table](#)

executeAndGuessTableName and executeAndGuessTableNameAndPrimaryKeys

When populating a rowset with an SQL query, you typically use the `execute()` method to run the query and read the data. The `WLCachedRowSet` implementation provides the `executeAndGuessTableName` and `executeAndGuessTableNameAndPrimaryKeys` methods that extend the `execute` method to also determine the associated table metadata.

The `executeAndGuessTableName` method parses the associated SQL and sets the table name for all columns as the first word following the SQL keyword `FROM`.

The `executeAndGuessTableNameAndPrimaryKeys` method parses the SQL command to read the table name. It then uses the `java.sql.DatabaseMetaData` to determine the table's primary keys.

Note:

These methods rely on support in the DBMS or JDBC driver. They do not work with all DBMSs or all JDBC drivers.

Setting Table and Primary Key Information Using the MetaData Interface

You can also choose to manually set the table and primary key information using the `WLRowSetMetaData` interface.

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();  
// Sets one table name for all columns  
metaData.setTableName("employees");
```

or

```
metaData.setTableName("e_id", "employees");  
metaData.setTableName("e_name", "employees");
```

You can also use the `WLRowSetMetaData` interface to identify primary key columns.

```
metaData.setPrimaryKeyColumn("e_id", true);
```

See [weblogic.jdbc.rowset.WLRowSetMetaData](#) in *Java API Reference for Oracle WebLogic Server*.

Setting the Write Table

The `WLRowSetMetaData` interface includes the `setWriteTableName` method to indicate the only table that should be updated or deleted. This is typically used when a rowset is populated with a join from multiple tables, but the rowset should only update one table. Any column that is not from the write table is marked as read-only.

For instance, a rowset might include a join of orders and customers. The write table could be set to orders. If `deleteRow` were called, it would delete the order row, but not delete the customer row.



Note:

JSR-114 provides the `CachedRowSet.setTableName` (see [http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/CachedRowSet.html#setTableName\(java.lang.String\)](http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/CachedRowSet.html#setTableName(java.lang.String))) that provides the same functionality as the WebLogic `CachedRowSetMetaData.setWriteTableName` method. Calling either method marks those columns that do NOT belong to the write table as read-only. WebLogic also provides the `CachedRowSetMetaData.setTableName` method which is used to map which table a column belongs to. When setting the write table using `setTableName`, be careful to implement the method using the appropriate API for your application.

RowSets and Transactions

Most database or JDBC applications use transactions, and RowSets support transactions, including JTA transactions.

The common use case is to populate the RowSet in Transaction 1. Transaction 1 commits, and there are no database or application server locks on the underlying data.

The RowSet holds the data in-memory, and it can be modified or shipped over the network to a client. When the application wishes to commit the changes to the database, it starts Transaction 2 and calls the RowSet's `acceptChanges` method. It then commits Transaction 2.

Integrating with JTA Global Transactions

The EJB container and the `UserTransaction` interface start transactions with the JTA transaction manager. The RowSet operations can participate in this transaction. To participate in the JTA transaction, the RowSet *must* use a transaction-aware DataSource (`TxDataSource`). The DataSource can be configured in the WebLogic Server console.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet aborts the global JTA transaction. The application will typically re-read the data and process the update again in a new transaction.

Behavior of Rowsets Using Global Transactions

In the case of a failure or rollback, the data is rolled back from the database, but is not rolled back from the rowset. Before proceeding you should do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.
- Create a new rowset with current data.

Using Local Transactions

If a JTA global transaction is not being used, the RowSet uses a local transaction. It first calls `setAutoCommit(false)` on the connection, then it issues all of the SQL statements, and finally it calls `connection.commit()`. This attempts to commit the local transaction. This method should *not* be used when trying to integrate with a JTA transaction that was started by the EJB or JMS containers.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet rolls back the local transaction. In this case, none of the SQL issued in `acceptChanges` will commit to the database.

Behavior of Rowsets Using Local Transactions

This section provides information on the behavior of rowsets in failed local transactions. The behavior depends on the type of connection object:

Calling `connection.commit`

In this situation, the connection object is not created by the rowset and initiates a local transaction by calling `connection.commit`. If the transaction fails or if the connection calls `connection.rollback`, the data is rolled back from the database, but is not rolled back in the rowset. Before proceeding, you must do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.
- Create a new rowset with current data.

Calling `acceptChanges`

In this situation, the rowset creates its own connection object and uses it to update the data in rowset by calling `acceptChanges`. In the case of failure or if the rowset calls `connection.rollback`, the data is be rolled back from the rowset and also from the database.

Reusing a WebLogic RowSet After Completing a Transaction

In many cases, after you synchronize changes in the rowset with the database, you may want to continue to use the rowset with its current data, which can improve application performance by reducing the number of database round trips. However, to reuse the rowset and its data, WebLogic Server needs to make sure that any transaction in which the rowset participates has completed before allowing you to make further changes to the data.

If you use a rowset in a local transaction and if `autocommit=true` is set on the connection object before rowset data changes are synchronized with the database, you can reuse the rowset with its current data after synchronizing the data because the `autocommit` setting forces the local transaction to complete immediately. WebLogic Server can be sure that the local transaction is complete before any further changes are made to the rowset.

WebLogic Server cannot automatically be sure that all transactions are complete if you use a rowset in either of the following scenarios:

- In a global transaction
- In a local transaction using a connection object with `autocommit=false` to synchronize data changes with the database

With either of these conditions, before you can reuse a rowset with its current data, after calling `acceptChanges()` to synchronize your changes with the database, you must call `javax.sql.rowset.CachedRowSet.commit()` instead of `tx.commit()` or `java.sql.Connection.commit()` to commit the transaction. The `CachedRowSet.commit()` method wraps the `Connection.commit()` method and enables WebLogic Server to ensure that the transaction is complete before allowing changes to the rowset.

FilteredRowSets

Learn how to use standard `FilteredRowSets` with WebLogic Server.

- [FilteredRowSet Characteristics](#)
- [Special Programming Considerations](#)
- [FilteredRowSet Code Example](#)
- [Importing Classes and Interfaces for FilteredRowSets](#)
- [Creating a FilteredRowSet](#)
- [Setting FilteredRowSet Properties](#)
- [Database Connection Options for a FilteredRowSet](#)
- [Populating a FilteredRowSet](#)

- [Setting FilteredRowSet MetaData](#)
- [Setting the Filter for a FilteredRowSet](#)
- [Working with Data in a FilteredRowSet](#)

FilteredRowSet Characteristics

A FilteredRowSet enables you to work with a subset of cached rows and change the subset of rows while disconnected from the database. A filtered rowset is simply a cached rowset in which only certain rows are available for viewing, navigating, and manipulating. FilteredRowSets have the following characteristics:

- The rows available are determined by a `javax.sql.rowset.Predicate` object supplied by the application and set with the `setFilter()` method.
 - The Predicate object must implement the `javax.sql.rowset.Predicate` interface. The Predicate interface includes the `public boolean evaluate(ResultSet rs)` method, which evaluates each row in the rowset
 - If the method returns `true`, the row is available and visible.
 - If the method returns `false`, the row is not available or visible.
- See [Setting the Filter for a FilteredRowSet](#).
- WebLogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which is an implementation of the `javax.sql.rowset.Predicate` interface that you can use to define a filter for a FilteredRowSet using SQL-like WHERE clause syntax. See [SQLPredicate, a SQL-Style RowSet Filter](#).

Special Programming Considerations

- [RowSet Filters are Not Cumulative](#)
- [No Pending Changes Before Setting or Changing a Filter](#)

RowSet Filters are Not Cumulative

Current behavior of WebLogic implementation of a FilteredRowSet is that when you set a filter for the second time on a FilteredRowSet, the new filter *replaces* the old filter. JSR-114 is not clear on this point. The reference implementation does not behave the same way, it further filters the filtered rows in the rowset. You can accomplish the same effect by changing the second filter to filter on all necessary criteria.

No Pending Changes Before Setting or Changing a Filter

If you have pending changes in a rowset before you set or change the rowset filter, you must either accept the changes (call `acceptChanges()`) or restore the rowset data to its pre-changed state (call `restoreOriginal()`). WebLogic Server considers navigating within a rowset to be indicative of a possible change and requires you to call either one of these methods before allowing you to change the rowset filter. Note that `acceptChanges()` includes a round-trip to the database, whereas `restoreOriginal()` does not.

FilteredRowSet Code Example

The following example shows how to create a cached rowset and then apply and change a filter using the WebLogic Server SQLPredicate.

Example 9-2 FilteredRowSet Code Example

```
import javax.sql.rowset.FilteredRowSet;
import javax.sql.rowset.RowSetFactory;
import weblogic.jdbc.rowset.SQLPredicate;
public class FilteredRowSetDemo {
public static void main (String[] args) {
//DESIGNING lifecycle stage - Create the rowset and set properties
try {
//Create a RowSetFactory instance and from the factory,
//create a FilteredRowSet.
RowSetFactory rsfact =
RowSetProvider.newFactory("weblogic.jdbc.rowset.JdbcRowSetFactory",null);
FilteredRowSet rs = rsfact.createFilteredRowSet();
//Set database access through a DataSource.
//See Database Connection Options for more options.
rs.setDataSourceName(examples-datasource-demoPool);
rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
PHONE, EMAIL FROM PHYSICIAN WHERE ID>?");
//CONFIGURE QUERY lifecycle operation - set values for query parameters.
rs.setInt(1, 0);
//POPULATING lifecycle stage - Execute the command to populate the rowset
rs.execute();
}
//CONFIGURING METADATA - Populate first, then set MetaData, including KeyColumns
rs.setKeyColumns(new int[] { 1 });
while (rs.next ())
//NAVIGATE operations put the rowset in the MANIPULATING lifecycle stage
{
System.out.println ("ID: " +rs.getInt (1));
System.out.println ("FIRST_NAME: " +rs.getString (2));
System.out.println ("MIDDLE_NAME: " +rs.getString (3));
System.out.println ("LAST_NAME: " +rs.getString (4));
System.out.println ("PHONE: " +rs.getString (5));
System.out.println ("EMAIL: " +rs.getString (6));
}
}
//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
rs.restoreOriginal();
//S E T F I L T E R
//use SQLPredicate class to create a SQLPredicate object,
//then pass the object in the setFilter method to filter the RowSet.
SQLPredicate filter = new SQLPredicate("ID >= 103");
rs.setFilter(filter);
System.out.println("Filtered data: ");
while (rs.next ())
{
System.out.println ("ID: " +rs.getInt (1));
System.out.println ("FIRST_NAME: " +rs.getString (2));
System.out.println ("MIDDLE_NAME: " +rs.getString (3));
System.out.println ("LAST_NAME: " +rs.getString (4));
System.out.println ("PHONE: " +rs.getString (5));
```



```

        System.out.println ("EMAIL: " +rs.getString (6));
        System.out.println (" ");
    }
//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING lifecycle stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
    rs.restoreOriginal();
//C H A N G I N G   F I L T E R
    SQLPredicate filter2 = new SQLPredicate("ID <= 103");
    rs.setFilter(filter2);
    System.out.println("Filtered data: ");
    while (rs.next ())
    {
        System.out.println ("ID: " +rs.getInt (1));
        System.out.println ("FIRST_NAME: " +rs.getString (2));
        System.out.println ("MIDDLE_NAME: " +rs.getString (3));
        System.out.println ("LAST_NAME: " +rs.getString (4));
        System.out.println ("PHONE: " +rs.getString (5));
        System.out.println ("EMAIL: " +rs.getString (6));
        System.out.println (" ");
    }
//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING lifecycle stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
    rs.restoreOriginal();
//R E M O V I N G   F I L T E R
    rs.setFilter(null);
    while (rs.next ())
    {
        System.out.println ("ID: " +rs.getInt (1));
        System.out.println ("FIRST_NAME: " +rs.getString (2));
        System.out.println ("MIDDLE_NAME: " +rs.getString (3));
        System.out.println ("LAST_NAME: " +rs.getString (4));
        System.out.println ("PHONE: " +rs.getString (5));
        System.out.println ("EMAIL: " +rs.getString (6));
        System.out.println (" ");
    }
    rs.close();
}
}

```

Importing Classes and Interfaces for FilteredRowSets

For standard FilteredRowSets, you must import the following classes:

```

javax.sql.rowset.FilteredRowSet;
javax.sql.rowset.RowSetFactory;

```

The preceding code example also uses the `weblogic.jdbc.rowset.SQLPredicate` class to create a filter. In your application, you can use the `weblogic.jdbc.rowset.SQLPredicate` class or you can create your own filter class. See [Setting the Filter for a FilteredRowSet](#).

Creating a FilteredRowSet

Rowsets are created from a factory interface. To create a FilteredRowSet with WebLogic Server, follow these main steps:

1. Create a `RowSetFactory` instance, which serves as a factory to create rowset objects for use in your application. For example:

```
RowSetFactory rsfact =  
RowSetProvider.newFactory("weblogic.jdbc.rowset.JdbcRowSetFactory", null);
```

2. Create a `javax.sql.rowset.FilteredRowSet` object. For example:

```
FilteredRowSet rs = rsfact.createCachedRowSet();
```

Setting FilteredRowSet Properties

Property options for a `FilteredRowSet` are the same as those for a `CachedRowSet`. See [Setting CachedRowSet Properties](#).

Database Connection Options for a FilteredRowSet

Database connection options for a `FilteredRowSet` are the same as those for a `CachedRowSet`. See [Database Connection Options](#).

Populating a FilteredRowSet

Data population options for a `FilteredRowSet` are the same as those for a `CachedRowSet`. See [Populating a CachedRowSet](#).

Setting FilteredRowSet MetaData

In some cases, you may need to set metadata for the rowset in order to synchronize data changes in the rowset with data in the database. See [RowSet MetaData Settings for Database Updates](#).

Setting the Filter for a FilteredRowSet

To filter the rows in a `FilteredRowSet`, you must call the `setFilter` method and pass a predicate (filter) object as a parameter of the method. The predicate object is an instance of a class that implements the `javax.sql.rowset.Predicate` interface. With the WebLogic implementation of `FilteredRowSets`, you can define your own filter or use an instance of the `weblogic.jdbc.rowset.SQLPredicate` class.

User-Defined RowSet Filter

When defining the filter for a `FilteredRowSet`, you follow these main steps:

1. Define a class that implements the `javax.sql.rowset.Predicate` interface with the filtering *behavior* you plan to use, such as limiting displayed rows to rows with a value in a particular column. For example, you may want to limit displayed rows based on a range of values for the ID column. The class you define would include logic to filter values for the ID column
2. Create an instance of the class (a filter) to specify the filtering *criteria* that you want to use. For example, you may want to see only rows with values in the ID column between 100 and 199.
3. Call `rowset.setFilter()` and pass the class as a parameter of the method.

Example 9-3 Filter Class that Implements javax.sql.rowset.Predicate

```

package examples.jdbc.rowsets;
import javax.sql.rowset.Predicate;
import javax.sql.rowset.CachedRowSet;
import javax.sql.RowSet;
import java.sql.SQLException;
public class SearchPredicate implements Predicate, java.io.Serializable {
    private boolean DEBUG = false;
    private String col = null;
    private String criteria = null;
    //Constructor to create case-insensitive column - value comparison.
    public SearchPredicate(String col, String criteria) {
        this.col = col;
        this.criteria = criteria;
    }
    public boolean evaluate(RowSet rs) {
        CachedRowSet crs = (CachedRowSet)rs;
        boolean bool = false;
        try {
            debug("evaluate(): "+crs.getString(col).toUpperCase()+" contains "+
                criteria.toUpperCase()+" = "+
                crs.getString(col).toUpperCase().contains(criteria.toUpperCase()));
            if (crs.getString(col).toUpperCase().contains(criteria.toUpperCase()))
                bool = true;
        } catch(Throwable t) {
            t.printStackTrace();
            throw new RuntimeException(t.getMessage());
        }
        return bool;
    }
    public boolean evaluate(Object o, String s) throws SQLException {
        throw new SQLException("String evaluation is not supported.");
    }
    public boolean evaluate(Object o, int i) throws SQLException {
        throw new SQLException("Int evaluation is not supported.");
    }
}

```

Example 9-4 Code to Set a Filter for a FilteredRowSet

```

SearchPredicate pred = new SearchPredicate(ROWSET_LASTNAME, lastName);
rs.setFilter(pred);

```

[Example 9-3](#) shows an example of a class that implements the `javax.sql.rowset.Predicate` interface. This example shows a class that enables you to create a filter that evaluates a case-insensitive version of the value in a column.

[Example 9-4](#) shows code to create an instance of the class, which determines the filter criteria, and then set the filter object as the filter for a `FilteredRowSet`.

WebLogic SQL-Style Filter

WebLogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which implements the `javax.sql.rowset.Predicate` interface. You can use the `SQLPredicate` class to define a filter using SQL-like WHERE clause syntax to filter rows in a rowset. For example:

```

SQLPredicate filter = new SQLPredicate("ID >= 103");
rs.setFilter(filter);

```

See [SQLPredicate](#), a SQL-Style RowSet Filter.

Working with Data in a FilteredRowSet

Working with data in a FilteredRowSet is much the same as working with data in a CachedRowSet, except that when you insert a row or update a row, the changes that you make must be within the filtering criteria so that the row will remain in the set of rows displayed. For example, if the filter on the rowset allowed only rows with an ID column value of less than 105 to be displayed, if you tried to insert a row with a value of 106 in the ID column or update an ID value to 106, that operation would fail and throw an SQLException.

For more details about working with data, see [Working with Data in a CachedRowSet](#).

WebRowSets

A WebRowSet is a cached rowset that can read and write a rowset in XML format.

WebRowSets have the following characteristics:

- Uses the `readXml(java.io.InputStream iStream)` method to populate the rowset from an XML source.
- Uses the `writeXml(java.io.OutputStream oStream)` method to write data and metadata in XML for use by other application components or to send to a remote client.
- The XML code used to populate the rowset or written from the rowset conforms to the standard WebRowSet XML Schema definition available at <http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/jdbc/webrowset.xsd>.

See <http://www.oracle.com/technetwork/java/javase/jdbc/index.html> and the Javadoc for the `javax.sql.rowset.WebRowSet` interface at <http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/WebRowSet.html>.



Note:

WebLogic Server supports two schemas for rowsets: one for the standard WebRowSet and one for the WLCachedRowSet, which was implemented before JSR-114 was finalized.

Special Programming Considerations

- The WebLogic WebRowSets implementation supports two XML schemas (and APIs): one for the standard WebRowSet specification (available at <http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/jdbc/webrowset.xsd>.) and one for the WLCachedRowSet, which was implemented before JSR-114 was finalized.
- If you are using only WebLogic Server rowsets, you can use either schema. The proprietary schema has more element types.
- To interact with other rowset implementations, you must use the standard schema.

JoinRowSets

A JoinRowSet is a number of disconnected RowSet objects joined together in a single rowset by a SQL JOIN.

JoinRowSets have the following characteristics:

- Each rowset added to the JoinRowSet must have a "match" column specified in the addRowSet method used to add the rowset to the JoinRowSet. For example:

```
addRowSet(javax.sql.RowSet[] rowset, java.lang.String[] columnName);
```

- You can set the join type using setJoinType method. The following join types are supported:

```
CROSS_JOIN
FULL_JOIN
INNER_JOIN
LEFT_OUTER_JOIN
RIGHT_OUTER_JOIN
```

- Enables you to join data while disconnected from the database.
- JoinRowSets are for read-only use. JoinRowSets cannot be used to update data in the database.
- Match columns in a JoinRowSet are limited to four data types: Number, Boolean, Date, and String. [Table 9-1](#) provides more details about data types allowed for a match column in a JoinRowSet.

Table 9-1 Data Types Allowed for Match Columns

Left Data Type in the Join	Allowed Right Data Types in the Join
Number	Number String
Boolean	Boolean String
Date	Date String
String	String Number Boolean Date

For more information about JoinRowSets, see the Javadoc for the `javax.sql.rowset.Joinable` (<http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/Joinable.html>) and `JoinRowSet` interfaces (<http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/JoinRowSet.html>).

JDBCRowSets

A `JDBCRowSet` is a wrapper around a `ResultSet` object that enables you to use the result set as a JavaBeans component. Note that a `JDBCRowSet` is a connected rowset. All other rowset types are disconnected rowsets.

See the Javadoc for the `javax.sql.rowset.JdbcRowSet` interface at <http://docs.oracle.com/javase/8/docs/api/javax/sql/rowset/JdbcRowSet.html>.

Handling SyncProviderExceptions with a SyncResolver

Learn about the steps for handling `SyncProviderException` with a `SyncResolver`. The `SyncProviderException` throws an error when it encounters violations in reading from or writing to the originating data source. The `SyncResolver` object may be used to examine and resolve each conflict in a row and then go to the next row with a conflict to repeat the procedure.

When you call `acceptChanges()` to propagate changes in a rowset to the database, WebLogic Server compares the original data in the rowset (data since the last synchronization) based on an optimistic concurrency policy with the data in the database. If it detects data changes, it throws a `javax.sql.rowset.spi.SyncProviderException`. By default, your application does not have to do anything, but the changes in the rowset will not be synchronized in the database. You can design your application to handle these exceptions and process the data changes as is suitable for your system.


Note:

For `javax.sql.rowset.CachedRowSets`, WebLogic Server compares all original values in all rows in the rowset with the corresponding rows in the database. For `weblogic.jdbc.rowset.WLCachedRowSet` or other WebLogic extended rowset types, WebLogic Server makes the data comparison based on the optimistic concurrency setting. See [Optimistic Concurrency Policies](#).

The main steps for handling a `SyncProviderException` are:

1. Catch the `javax.sql.rowset.spi.SyncProviderException`.
2. Get the `SyncResolver` object from the exception. See [Getting a SyncResolver Object](#).
3. Page through conflicts using `nextConflict()` or any other navigation method. See [Navigating in a SyncResolver Object](#).
4. Determine the correct value, then set it with `setResolvedValue()`, which sets the value in the rowset. See [Setting the Resolved Value for a RowSet Data Synchronization Conflict](#).
5. Repeat steps 3 and 4 for each conflicted value.
6. Call `rowset.acceptChanges()` on the rowset (not the `SyncResolver`) to synchronize changes with the database using the new resolved values. See [Synchronizing Changes](#).

For more details about SyncResolvers and the SyncProviderException, see the RowSets specification or the Javadoc for the SyncResolver interface.

 **Note:**

Before you begin to resolve the SyncProviderException, make sure that no other processes will update the data.

RowSet Data Synchronization Conflict Types

Table 9-2 lists the types of conflict scenarios that can occur when synchronizing data changes from a rowset to the database.

Table 9-2 Conflict Types When Synchronizing RowSet Changes in the Database

RowSet Data Change Type	Database Data Change Type	Notes
Update	Update	<p>Values in the same row in the rowset and database have changed. The syncresolver status is SyncResolver.UPDATE_ROW_CONFLICT.</p> <p>Your application may need to supply logic to resolve the conflict or may need to present the new data to the user.</p>
Update	Delete	<p>Values in the row in the rowset have been updated, but the row has been deleted in the database. The syncresolver status is SyncResolver.UPDATE_ROW_CONFLICT.</p> <p>Your application may need to supply logic to decide whether to leave the row as deleted (as it is in the database) or to restore the row and persist changes from the rowset.</p> <ul style="list-style-type: none"> To leave the row as deleted, revert the changes to the row in the rowset. To restore the row with changes, insert a new row with the desired values. <p>Note that if the row is deleted in the database, there is no conflict value. When you call <code>getConflictValue()</code>, WebLogic Server throws a <code>weblogic.jdbc.rowset.RowNotFoundException</code>.</p>
Delete	Update	<p>The row has been deleted in the rowset, but the row has been updated in the database. The syncresolver status is SyncResolver.DELETE_ROW_CONFLICT.</p> <p>Your application may need to supply logic to decide whether to delete the row (as it is in the rowset) or to keep the row and persist changes currently in the database.</p> <p>Note that in this scenario, all values in the row will be conflicted values. To keep the row with the current values in the database, call <code>setResolvedValue</code> to set the resolved value for each column in the row to the current value in the database. To proceed with the delete, call <code>syncprovider.deleteRow()</code>.</p>

Table 9-2 (Cont.) Conflict Types When Synchronizing RowSet Changes in the Database

RowSet Data Change Type	Database Data Change Type	Notes
Delete	Delete	<p>The row has been deleted in the rowset and has been deleted in the database by another process. The syncresolver status is SyncResolver.DELETE_ROW_CONFLICT.</p> <p>To resolve the SyncProviderException, you must revert the delete operation on the row in the rowset.</p> <p>Note that there will be no conflict value (not null, either) for any column in the row. When you call <code>getConflictValue()</code>, WebLogic Server throws a <code>weblogic.jdbc.rowset.RowNotFoundException</code>.</p>
Insert	Insert	<p>If a row is inserted in the rowset and a row is inserted in the database, a primary key conflict may occur, in which case an SQL exception will be thrown. You cannot directly handle this conflict type using a SyncResolver because a SyncProviderException is not thrown.</p>

SyncResolver Code Example

[Example 9-5](#) shows an abbreviated example of how to use a SyncResolver to resolve conflicting values between the rowset and the database. This example checks the value for known column names in each row in the SyncResolver in which there is a conflict. Details about the example are explained in the sections that follow the example.

Example 9-5 SyncResolver Abbreviated Code Example

```
try {
    rs.acceptChanges();
} catch (SyncProviderException spex) {
    SyncResolver syncresolver = spex.getSyncResolver();
    while (syncresolver.nextConflict()) {
        int status = syncresolver.getStatus();
        int rownum = syncresolver.getRow();
        rs.absolute(rownum);
        //check for null in each column
        //write out the conflict
        //set resolved value to value in the db for this example
        //handle exception for deleted row in the database
        try {
            Object idConflictValue = syncresolver.getConflictValue("ID");
            if (idConflictValue != null) {
                System.out.println("ID value in db: " + idConflictValue);
                System.out.println("ID value in rowset: " + rs.getInt("ID"));
                syncresolver.setResolvedValue("ID", idConflictValue);
                System.out.println("Set resolved value to " + idConflictValue);
            }
            else {
                System.out.println("ID: NULL - no conflict");
            }
        } catch (RowNotFoundException e) {
            System.out.println("An exception was thrown when requesting a ");
        }
    }
}
```



```

        System.out.println("value for ID. This row was ");
        System.out.println("deleted in the database.");
    }
    . . .
    }
    try {
        rs.acceptChanges();
    } catch (Exception ignore2) {
    }
}

```

Getting a SyncResolver Object

To handle a `SyncProviderException`, you can catch the exception and get a `SyncResolver` object from it. For example:

```

try {
    rowset.acceptChanges();
} catch (SyncProviderException spex) {
    SyncResolver syncresolver = spex.getSyncResolver();
    . . .
}

```

A `SyncResolver` is a rowset that implements the `SyncResolver` interface. A `SyncResolver` object contains a row for every row in the original rowset. For values without a conflict, the value in the `SyncResolver` is null. For values with a conflict, the value is the current value in the database.

Navigating in a SyncResolver Object

With a `SyncResolver` object, you can page through all conflicts and set the appropriate value for each conflict value. The `SyncResolver` interface includes the `nextConflict()` and `previousConflict()` methods that you can use to navigate directly to the next row in the `SyncResolver` that has a conflict value other than null. Because a `SyncResolver` object is a rowset, you can also use all of the rowset navigation methods to move the cursor to any row in the `SyncResolver`. However, the `nextConflict()` and `previousConflict()` methods enable you to easily skip rows that do not contain conflict values.

After you move the cursor to a conflict row, you must check the value in each column with the `getConflictValue()` method to find the values in the database that conflict with the values in the rowset, and then compare values to determine how to handle the conflict. For rows with values that do not conflict, the return value is null. If the row was deleted in the database, there is no value to return, so an exception is thrown.

Note:

In the WebLogic rowsets implementation, a value conflict occurs if any value in a row in the database differs from the values read into the rowset when the rowset was created or when it was last synchronized.

An example of code to compare values in the rowset and database:

```
syncresolver.nextConflict()
for (int i = 1; i <= colCount; i++) {
    if (syncresolver.getConflictValue(i) != null) {
        rsValue = rs.getObject(i);
        resolverValue = syncresolver.getConflictValue(i);
        . . .
        // compare values in the rowset and SyncResolver to determine
        // which should be the resolved value (the value to persist)
    }
}
```

Setting the Resolved Value for a RowSet Data Synchronization Conflict

To set the appropriate value to persist in the database, you call `setResolvedValue()`. For example:

```
syncresolver.setResolvedValue(i, resolvedValue);
```

The `setResolvedValue()` method makes the following changes:

- Sets the value to persist in the database. That is, it sets the current value in the rowset. When changes are synchronized, the new value will be persisted to the database.
- Changes the original value for the rowset data to the current value in the database. The original value was the value since the last synchronization. After calling `setResolvedValue()`, the original value becomes the current value in the database.
- Changes the WHERE clause in the synchronization call so that updates are made to appropriate rows in the database.

Synchronizing Changes

After resolving conflicting values in the SyncResolver, you must synchronize your changes with the database. To do that, you call `rowset.acceptChanges()`. again. The `acceptChanges()` call closes the SyncResolver object and releases locks on the database after the synchronization completes.

WLCachedRowSets

A `WLCachedRowSet` is an extension of `CachedRowSets`, `FilteredRowSets`, `WebRowSets`, and `SortedRowSets`.

`WLCachedRowSet` has the following characteristics:

- In the WebLogic Server RowSets implementation, all rowsets originate as a `WLCachedRowSet`. `WLCachedRowSet` can be interchangeably used as any of the standard rowset types that it extends.
- `WLCachedRowSet` include convenience methods that help make using rowsets easier and also include methods for setting optimistic concurrency options and data synchronization options.
- It may not be possible to read or update an `SQLXML` datatype object. The JDBC 4.0 specification does not require vendors to make `SQLXML` objects readable after

they have been set. Once WebLogic Server sets the value for an `SQLXML` datatype object, it cannot be read or updated.

See [weblogic.jdbc.rowset.WLCachedRowSet](#) in *Java API Reference for Oracle WebLogic Server* interface.

SharedRowSets

Rowsets can be used by a single thread. They cannot be shared by multiple threads. A `SharedRowSet` extends `CachedRowSets` so that additional `CachedRowSets` can be created for use in other threads based on the data in an original `CachedRowSet`.

`SharedRowSets` have the following characteristics:

- Each `SharedRowSet` is a shallow copy of the original rowset (with references to data in the original rowset instead of a copy of the data) with its own context (cursor, filter, sorter, pending changes, and sync provider).
- When data changes from any of the `SharedRowSets` are synchronized with the database, the base `CachedRowSet` is updated as well.
- Using `SharedRowSets` can increase performance by reducing the number of database round-trips required by an application.

To create a `SharedRowSet`, you use the `createShared()` method in the `WLCachedRowSet` interface and cast the result as a `WLCachedRowSet`. For example:

```
WLCachedRowSet sharedrowset = (WLCachedRowSet)rowset.createShared();
```

SortedRowSets

A `SortedRowSet` extends `CachedRowSets` so that rows in a `CachedRowSet` can be sorted based on the `Comparator` object provided by the application.

`SortedRowSets` have the following characteristics:

- Sorting is set in a way similar to way filtering is set for a `FilteredRowSet`, except that sorting is based on a `java.util.Comparator` object instead of a `javax.sql.rowset.Predicate` object:
 1. The application creates a `Comparator` object with the desired sorting behavior.
 2. The application then sets the sorting criteria with the `setSorter(java.util.Comparator)` method.
- Sorting is done in memory rather than depending on the database management system for sort processing. Using `SortedRowSets` can increase application performance by reducing the number of database round-trips.
- WebLogic Server provides the `SQLComparator` object, which implements `java.util.Comparator`. You can use it to sort rows in a `SortedRowSet` by passing the list of columns that you want use as sorting criteria. For example:

```
rs.setSorter(new  
weblogic.jdbc.rowset.SQLComparator("columnA,columnB,columnC"));
```

See the following interface and class in *Java API Reference for Oracle WebLogic Server*:

- [weblogic.jdbc.rowset.SortedRowSet](#) interface

- [weblogic.jdbc.rowset.SQLComparator](#) class

SQLPredicate, a SQL-Style RowSet Filter

The SQLPredicate class is used to define a filter for a FilteredRowSet using SQL-like WHERE clause syntax.

- [What is SQLPredicate?](#)
- [SQLPredicate Grammar](#)
- [Code Example](#)

What is SQLPredicate?

WebLogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which is an implementation of the `javax.sql.rowset.Predicate` interface. You can use the `SQLPredicate` class to define a filter for a `FilteredRowSet` using SQL-like WHERE clause syntax.

SQLPredicate Grammar

The SQLPredicate class borrows its grammar from the JMS selector grammar, which is very similar to the grammar for an SQL select WHERE clause.

Some important notes:

- When referencing a column, you must use the column name; you cannot use column index number.
- The grammar supports the use of operators and mathematical operations, for example:

```
(colA + ColB) >=100.
```
- In constructing the WHERE clause, you can use simple datatypes only, including:
 - String
 - Int
 - Boolean
 - Float
- Complex data types are *not* supported:
 - Array
 - BLOB
 - CLOB
 - Date

Code Example

```
//S E T   F I L T E R
//use SQLPredicate class to create a SQLPredicate object,
//then pass the object in the setFilter method to filter the RowSet.
```

```
SQLPredicate filter = new SQLPredicate("ID >= 103");  
rs.setFilter(filter);
```

See [weblogic.jdbc.rowset.SQLPredicate](#) class in *Java API Reference for Oracle WebLogic Server*.

Optimistic Concurrency Policies

With optimistic concurrency, RowSets work on the assumption that multiple users are unlikely to change the same data at the same time. Therefore, as part of the disconnected rowset model, the rowset does not lock database resources.

In most cases, populating a rowset with data and updating the database occur in separate transactions. The underlying data in the database can change in the time between the two transactions. The WebLogic Server rowset implementation (WLCachedRowSet) uses optimistic concurrency control to ensure data consistency.

With optimistic concurrency, RowSets work on the assumption that multiple users are unlikely to change the same data at the same time. Therefore, as part of the disconnected rowset model, the rowset does not lock database resources. However, before writing changes to the database, the rowset must check to make sure that the data to be changed in the database has not already changed since the data was read into the rowset.

The UPDATE and DELETE statements issued by the rowset include WHERE clauses that are used to verify the data in the database against what was read when the rowset was populated. If the rowset detects that the underlying data in the database has changed, it issues an `OptimisticConflictException`. The application can catch this exception and determine how to proceed. Typically, applications will refresh the updated data and present it to the user again.

The WLCachedRowSet implementation offers several optimistic concurrency policies that determine what SQL the rowset issues to verify the underlying database data:

- VERIFY_READ_COLUMNS
- VERIFY_MODIFIED_COLUMNS
- VERIFY_SELECTED_COLUMNS
- VERIFY_NONE
- VERIFY_AUTO_VERSION_COLUMNS
- VERIFY_VERSION_COLUMNS

To illustrate the differences between these policies, we will use an example that uses the following:

- A very simple employees table with 3 columns:

```
CREATE TABLE employees (  
    e_id integer primary key,  
    e_salary integer,  
    e_name varchar(25)  
);
```

- A single row in the table:

```
e_id = 1, e_salary = 10000, and e_name = 'John Smith'
```

In the example for each of the optimistic concurrency policies listed below, the rowset will read this row from the employees table and set John Smith's salary to 20000. The example will then show how the optimistic concurrency policy affects the SQL code issued by the rowset.

VERIFY_READ_COLUMNS

The default rowset optimistic concurrency control policy is `VERIFY_READ_COLUMNS`. When the rowset issues an `UPDATE` or `DELETE`, it includes all columns that were read from the database in the `WHERE` clause. This verifies that the value in all columns that were initially read into the rowset have not changed.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000 AND e_name = 'John Smith';
```

VERIFY_MODIFIED_COLUMNS

The `VERIFY_MODIFIED_COLUMNS` policy only includes the primary key columns and the updated columns in the `WHERE` clause. It is useful if your application only cares if its updated columns are consistent. It does allow your update to commit if columns that have not been updated have changed since the data has been read.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000
```

The `e_id` column is included since it is a primary key column. The `e_salary` column is a modified column so it is included as well. The `e_name` column was only read so it is not verified.

VERIFY_SELECTED_COLUMNS

The `VERIFY_SELECTED_COLUMNS` includes the primary key columns and columns you specify in the `WHERE` clause.

```
WRowSetMetaData metaData = (WRowSetMetaData) rowSet.getMetaData();
metaData.setOptimisticPolicy(WRowSetMetaData.VERIFY_SELECTED_COLUMNS);
// Only verify the e_salary column
metaData.setVerifySelectedColumn("e_salary", true);

metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
  WHERE e_id = 1 AND e_salary=10000
```

The `e_id` column is included since it is a primary key column. The `e_salary` column is a selected column so it is included as well.

VERIFY_NONE

The VERIFY_NONE policy only includes the primary key columns in the WHERE clause. It does not provide any additional verification on the database data.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000 WHERE e_id = 1
```

VERIFY_AUTO_VERSION_COLUMNS

The VERIFY_AUTO_VERSION_COLUMNS includes the primary key columns as well as a separate version column that you specify in the WHERE clause. The rowset will also automatically increment the version column as part of the update. This version column must be an integer type. The database schema must be updated to include a separate version column (e_version). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.  
    VERIFY_AUTO_VERSION_COLUMNS);  
  
metaData.setAutoVersionColumn("e_version", true);  
  
metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000, e_version = 2  
WHERE e_id = 1 AND e_version = 1
```

The e_version column is automatically incremented in the SET clause. The WHERE clause verified the primary key column and the version column.

VERIFY_VERSION_COLUMNS

The VERIFY_VERSION_COLUMNS has the rowset check the primary key columns as well as a separate version column. The rowset does not increment the version column as part of the update. The database schema must be updated to include a separate version column (e_version). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.VERIFY_VERSION_COLUMNS);  
metaData.setVersionColumn("e_version", true);  
metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000  
WHERE e_id = 1 AND e_version = 1
```

The WHERE clause verifies the primary key column and the version column. The rowset does not increment the version column so this must be handled by the database. Some databases provide automatic version columns that increment when the row is updated. It is also possible to use a database trigger to handle this type of update.

Optimistic Concurrency Control Limitations

The Optimistic policies only verify UPDATE and DELETE statements against the row they are changing. Read-only rows are not verified against the database.

Most databases do not allow BLOB or CLOB columns in the WHERE clause so the rowset never verifies BLOB or CLOB columns.

When multiple tables are included in the rowset, the rowset only verifies tables that have been updated.

Choosing an Optimistic Policy

The default VERIFY_READ_COLUMNS provides a strong-level of consistency at the expense of some performance. Since all columns that were initially read must be sent to the database and compared in the database, there is some additional overhead to this policy. VERIFY_READ_COLUMNS is appropriate when strong levels of consistency are needed, and the database tables cannot be modified to include a version column.

The VERIFY_SELECTED_COLUMNS is useful when the developer needs complete control over the verification and wants to use application-specific knowledge to fine-tune the SQL.

The VERIFY_AUTO_VERSION_COLUMNS provides the same level of consistency as VERIFY_READ_COLUMNS but only has to compare a single integer column. This policy also handles incrementing the version column so it requires a minimal amount of database setup.

The VERIFY_VERSION_COLUMNS is recommended for production systems that want the highest level of performance and consistency. Like VERIFY_AUTO_VERSION_COLUMNS, it provides a high level of consistency while only incurring a single column comparison in the database. VERIFY_VERSION_COLUMNS requires that the database handle incrementing the version column. Some databases provide a column type that automatically increments itself on updates, but this behavior can also be implemented with a database trigger.

The VERIFY_MODIFIED_COLUMNS and VERIFY_NONE decrease the consistency guarantees, but they also decrease the likelihood of an optimistic conflict. You should consider these policies when performance and avoiding conflicts outweigh the need for higher level of data consistency.

Performance Options

Learn about the RowSets performance options such as JDBC Batching and Group Deletes.

- [JDBC Batching](#)
- [Group Deletes](#)

JDBC Batching

The rowset implementation includes support for JDBC batch operations. Instead of sending each SQL statement individually to the JDBC driver, a batch sends a

collection of statements in one bulk operation to the JDBC driver. Batching is disabled by default, but it generally improves performance when large numbers of updates occur in a single transaction. It is worthwhile to benchmark with this option enabled and disabled for your application and database.

The `WLCachedRowSet` interface contains the methods `setBatchInserts(boolean)`, `setBatchDeletes(boolean)`, and `setBatchUpdates(boolean)` to control batching of INSERT, DELETE, and UPDATE statements.

 **Note:**

The `setBatchInserts`, `setBatchDeletes`, or `setBatchUpdates` methods must be called before the `acceptChanges` method is called.

Batching Limitations with and Oracle Database

Since the `WLCachedRowSet` relies on optimistic concurrency control, it needs to determine whether an update or delete command has succeeded or an optimistic conflict occurred. The `WLCachedRowSet` implementation relies on the JDBC driver to report the number of rows updated by a statement to determine whether a conflict occurred or not. In the case where 0 rows were updated, the `WLCachedRowSet` knows that a conflict did occur.

Oracle JDBC drivers return `java.sql.Statement.SUCCESS_NO_INFO` when batch updates are executed, so the rowset implementation cannot use the return value to determine whether a conflict occurred.

When the rowset detects that batching is used with an Oracle database, it automatically changes its batching behavior:

Batched inserts perform as usual since they are not verified.

Batched updates run as normal, but the rowset issues an extra SELECT query to check whether the batched update encountered an optimistic conflict.

Batched deletes use group deletes since this is more efficient than executing a batched delete followed by a SELECT verification query.

Group Deletes

When multiple rows are deleted, the rowset would normally issue a DELETE statement for each deleted row. When group deletes are enabled, the rowset issues a single DELETE statement with a WHERE clause that includes the deleted rows.

For instance, if we were deleting 3 employees from our table, the rowset would normally issue:

```
DELETE FROM employees WHERE e_id = 3 AND e_version = 1;
DELETE FROM employees WHERE e_id = 4 AND e_version = 3;
DELETE FROM employees WHERE e_id = 5 AND e_version = 10;
```

When group deletes are enabled, the rowset issues:

```
DELETE FROM employees
WHERE e_id = 3 AND e_version = 1 OR
```

```
e_id = 4 AND e_version = 3 OR  
e_id = 5 AND e_version = 10;
```

You can use the `WRowSetMetaData.setGroupDeleteSize` to determine the number of rows included in a single `DELETE` statement. The default value is 50.

10

Troubleshooting JDBC

Learn about common issues such as problems with Oracle Database on UNIX, thread-related problems on UNIX and so on.

- [Problems with Oracle Database on UNIX](#)
- [Thread-related Problems on UNIX](#)
- [Closing JDBC Objects](#)
- [Using Microsoft SQL Server with Nested Triggers](#)

Problems with Oracle Database on UNIX

If you have problems with an Oracle database running on Unix, check the threading model being used. When using Oracle drivers, WebLogic recommends that you use *native* threads. You can specify this by adding the `-native` flag when you start Java.

Thread-related Problems on UNIX

On UNIX, two threading models are available: green threads and native threads. You can determine what type of threads you are using by checking the environment variable called `THREADS_TYPE`.

For more information, read about the JDK for the Solaris operating environment at <http://www.oracle.com/technetwork/java/index.html>.

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

Some of the problems are related to the implementation of threads in the JVM for each operating system. Not all JVMs handle operating-system specific threading issues equally well. If you are using Oracle drivers, use *native* threads.

Closing JDBC Objects

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

Here is a general example:

Example 10-1 Closing a JDBC Object

```
try {  
  
    Driver d =  
        (Driver)Class.forName("oracle.jdbc.OracleDriver").newInstance();  
  
    Connection conn = d.connect("jdbc:weblogic:oracle:myserver",
```

```
        "scott", "tiger");

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

}

catch (Exception e) {

    // handle any exceptions as appropriate

}

finally {

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

}
```

Abandoning JDBC Objects

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

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

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

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

Using Microsoft SQL Server with Nested Triggers

Learn about the troubleshooting information when using nested triggers with some Microsoft SQL Server databases.

- [Exceeding the Nesting Level](#)
- [Using Triggers and EJBs](#)

For information on supported data bases and data base drivers, see the *Oracle Fusion Middleware Supported System Configurations* page at <http://www.oracle.com/technetwork/middleware/ias/downloads/fusion-certification-100350.html>.

Exceeding the Nesting Level

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

For example:

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

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

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

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

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

Results in the following error message:

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

To work around this issue, do the following:

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

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

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

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

Using Triggers and EJBs

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

For example:

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

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

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

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

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

To work around this issue, run the following script:

```
exec sp_configure 'show advanced options', 1
reconfigure with override
```

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