BEA WebLogic Server®

Configuring and Managing WebLogic JDBC

Version 10.0
Revised: March 30, 2007

Contents

1. Introduction and Roadmap

Document Scope and Audience 1-1
Guide to this Document 1-2
Related Documentation 1-2
JDBC Samples and Tutorials 1-3
Avitek Medical Records Application (MedRec) and Tutorials 1-3
JDBC Examples in the WebLogic Server Distribution. 1-3
Additional JDBC Examples Available for Download 1-3
New and Changed JDBC Features in This Release 1-4

2. Configuring WebLogic JDBC Resources

Understanding JDBC Resources in WebLogic Server 2-1
Ownership of Configured JDBC Resources 2-3
JDBC Configuration Files 2-3
JDBC System Modules 2-4
JDBC Application Modules 2-6
JDBC Module File Naming Requirements 2-7
JDBC Modules in Versioned Applications 2-7
JDBC Schema 2-8
JMX and WLST Access for JDBC Resources 2-8
JDBC MBeans for System Resources 2-8
JDBC Management Objects in the Java EE Management Model (JSR-77 Support) . 2-9

Using WLST to Create JDBC System Resources 2-10
How to Modify and Monitor JDBC Resources 2-13
Best Practices when Using WLST to Configure JDBC 2-13
Overview of Clustered JDBC 2-14

3. Configuring JDBC Data Sources

Understanding JDBC Data Sources 3-1
Creating a JDBC Data Source 3-2
General Data Source Options 3-3
Selecting a JDBC Driver 3-3
JDBC Data Source Names 3-4
Binding a Data Source to the JNDI Tree with Multiple Names 3-4
Transaction Options 3-4
Enabling Support for Global Transactions with a Non-XA JDBC Driver 3-5
Understanding the Logging Last Resource Transaction Option 3-6
Advantages to Using the Logging Last Resource Optimization 3-8
Enabling the Logging Last Resource Transaction Optimization 3-8
Programming Considerations and Limitations for LLR Data Sources 3-8
Administrative Considerations and Limitations for LLR Data Sources 3-10
Understanding the Emulate Two-Phase Commit Transaction Option 3-11
Limitations and Risks When Emulating Two-Phase Commit Using a Non-XA Driver 3-12
Connection Pool Features 3-13
Selecting a JDBC Driver 3-14
Enabling JDBC Driver-Level Features 3-15
Initializing Database Connections with SQL Code 3-15
Setting Database Security Credentials 3-15
Types of Data Source Pools 3-16

Using a User Name/Password 3-16
Set Client ID On Connection 3-17
Identity-based Connection Pooling 3-17
Tuning Data Source Connection Pool Options 3-19
Increasing Performance with the Statement Cache 3-20
Statement Cache Algorithms. 3-20
Statement Cache Size 3-21
Usage Restrictions for the Statement Cache 3-21
Connection Testing Options for a Data Source 3-23
Database Connection Testing Semantics. 3-24
Database Connection Testing Configuration Recommendations 3-27
Default Test Table Name. 3-27
Enabling Connection Creation Retries 3-28
Enabling Connection Requests to Wait for a Connection 3-29
Connection Reserve Timeout 3-29
Limiting the Number of Waiting Connection Requests 3-30
Automatically Recovering Leaked Connections 3-30
Avoiding Server Lockup with the Correct Number of Connections 3-30
Limiting Statement Processing Time with Statement Timeout 3-31
Using Pinned-To-Thread Property to Increase Performance 3-31
Changes to Connection Pool Administration Operations When PinnedToThread is Enabled. 3-32
Additional Database Resource Costs When PinnedToThread is Enabled 3-32
Deploying Data Sources on Servers and Clusters 3-33
Minimizing Server Startup Hang Caused By an Unresponsive Database. 3-33
Security for JDBC Data Sources 3-34
Setting Security Policies for JDBC Resources. 3-34
Security Roles for JDBC MBeans 3-34

JDBC Domain Configuration MBeans 3-34
JDBC System Module MBeans 3-35
JDBC Data Source Factories (Deprecated) 3-35

4. Configuring JDBC Multi Data Sources

Multi Data Source Features 4-2
Removing a Database Node 4-2
Adding a Database Node 4-3
Creating and Configuring Multi Data Sources 4-3
Choosing the Multi Data Source Algorithm 4-3
Failover 4-3
Load Balancing 4-4
Multi Data Source Fail-Over Limitations and Requirements 4-4
Test Connections on Reserve to Enable Fail-Over 4-4
No Fail-Over for In-Use Connections 4-5
Multi Data Source Failover Enhancements 4-5
Connection Request Routing Enhancements When a Data Source Fails 4-5
Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source 4-6
Enabling Failover for Busy Data Sources in a Multi Data Source 4-6
Controlling Multi Data Source Failover with a Callback 4-7
Callback Handler Requirements 4-7
Callback Handler Configuration 4-8
How It Works—Failover 4-8
Controlling Multi Data Source Failback with a Callback 4-9
How It Works—Failback 4-10
Deploying JDBC Multi Data Sources on Servers and Clusters 4-11

5. Using Third-Party JDBC Drivers with WebLogic Server

Third-Party JDBC Drivers Installed with WebLogic Server. 5-1

Setting the Environment for a Type-4 Third-Party JDBC Driver 5-2

Globalization Support for the Oracle 10g Thin Driver 5-3

Using the Oracle Thin Driver in Debug Mode 5-3

6. Monitoring WebLogic JDBC Resources

Viewing Runtime Statistics 6-2

Data Source Statistics 6-2

Prepared Statement Cache Statistics. 6-2

Collecting Profile Information. 6-2

Profile Types. 6-2

Connection Usage (PROFILE_TYPE_CONN_USAGE_STR) 6-3

Connection Reservation Wait (PROFILE_TYPE_CONN_RESV_WAIT_STR) 6-3

Connection Reservation Failed (PROFILE_TYPE_CONN_RESV_FAIL_STR)6-4

Connection Leak (PROFILE_TYPE_CONN_LEAK_STR) 6-4

Connection Last Usage (PROFILE_TYPE_CONN_LAST_USAGE_STR) . . . 6-4

Connection Multithreaded Usage (PROFILE_TYPE_CONN_MT_USAGE_STR) 6-5

Statement Cache Entry (PROFILE_TYPE_STMT_CACHE_ENTRY_STR) . . 6-5

Statements Usage (PROFILE_TYPE_STMT_USAGE_STR) 6-5

Accessing Diagnostic Data. 6-6

Callbacks for Monitoring Driver-Level Statistics 6-7

Debugging JDBC Data Sources. 6-7

Enabling Debugging. 6-7

Enable Debugging Using the Command Line. 6-7

Enable Debugging Using the WebLogic Server Administration Console 6-7

Enable Debugging Using the WebLogic Scripting Tool. 6-8

Changes to the config.xml File 6-10
JDBC Debugging Scopes 6-10
Request Dyeing 6-11

7. Managing WebLogic JDBC Resources

Testing Data Sources and Database Connections 7-1
Managing the Statement Cache for a Data Source 7-2
Clearing the Statement Cache for a Data Source 7-2
Clearing the Statement Cache for a Single Connection 7-2
Shrinking a Data Source 7-3
Resetting a Data Source 7-4
Suspending a Data Source 7-4
Resuming a Data Source 7-4
Shutting Down a Data Source 7-5
Starting a Data Source 7-5

A. Configuring JDBC Application Modules for Deployment

Packaging a JDBC Module with an Enterprise Application: Main Steps A-1
Creating Packaged JDBC Modules A-2
Creating a JDBC Data Source Module Using the Administration Console A-2
JDBC Packaged Module Requirements A-3
JDBC Application Module Limitations A-3
Creating a JDBC Data Source Module A-4
Creating a JDBC Multi Data Source Module A-5
Encrypting Database Passwords in a JDBC Module A-6
Deploying JDBC Modules to New Domains A-6
Application Scoping for a Packaged JDBC Module A-7
Referencing a JDBC Module in Java EE Descriptor Files A-7

Packaged JDBC Module References in weblogic-application.xml A-8
Packaged JDBC Module References in Other Descriptors A-9
Packaging an Enterprise Application with a JDBC Module A-10
Deploying an Enterprise Application with a JDBC Module A-10
Getting a Database Connection from a Packaged JDBC Module A-10

B. Using WebLogic Server with Oracle RAC

Overview of Oracle Real Application Clusters B-2
Oracle RAC Scalability with WebLogic Server B-3
Oracle RAC Availability with WebLogic Server B-3
Oracle RAC Load Balancing with WebLogic Server B-3
Oracle RAC Failover with WebLogic Server B-4
Environment B-4
Hardware Requirements B-4
WebLogic Server Cluster B-4
Oracle RAC Cluster B-4
Shared Storage B-5
Software Requirements B-5
Configuration Considerations for Oracle B-5
Configuring the Listener Process for Each Oracle RAC Instance B-6
Disabling Remote Listeners B-7
Configuration Options in WebLogic Server with Oracle RAC B-8
Choosing a WebLogic Server Configuration for Use with Oracle RAC B-8
Required JDBC Drivers B-9
Configuration Considerations for Failover B-9
Multi Data Source-Managed Failover B-9
Connect-Time Failover B-10
Delays During Failover B-10

Failure Handling Walkthrough for Global Transactions. B-12
Using Multi Data Sources with Oracle RAC. B-13
Attributes of a Multi Data Source. B-14
Using Multi Data Sources with Global Transactions. B-14
Rules for Data Sources within a Multi Data Source Using Global TransactionsB-15
Required Attributes of Data Sources within a Multi Data Source Using Global Transactions. B-15
Sample Configuration Code B-16
Using Multi Data Sources without Global Transactions B-18
Attributes of Data Sources within a Multi Data Source Not Using Global Transactions. B-18
Sample Configuration Code B-19
Using Connect-Time Failover with Oracle RAC. B-21
Using Connect-Time Failover without Global Transactions B-23
Attributes of a Connect-Time Failover Configuration without Global Transactions B-23
Sample Configuration Code B-23
Using Fast Connection Failover. B-25
Required JDBC Driver Configuration for use with Oracle Fast Connection Failover B-25
XA Considerations and Limitations with Oracle RAC. B-25
Required JDBC Driver Configuration for Use with XA B-25
Oracle 9i RAC XA Requirements B-25
A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the RAC Cluster B-26
Transaction IDs Must Be Unique Within the RAC Cluster B-26
Known Limitations When Using Oracle 9i RAC with WebLogic Server B-26

Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions B-27
Potential for Data Deadlocks in Some Failure Scenarios B-28
Potential for Transactions Completed Out of Sequence B-28
Known Issue Occurring After Database Server Crash B-28
JDBC Store Recovery with Oracle RAC B-28
Configuring a JDBC Store for Use with Oracle RAC B-29
Automatic Retry B-29

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Configuring and Managing WebLogic JDBC*.

- "Document Scope and Audience" on page 1-1
- "Guide to this Document" on page 1-2
- "Related Documentation" on page 1-2
- "JDBC Samples and Tutorials" on page 1-3
- "New and Changed JDBC Features in This Release" on page 1-4

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.

This document does not address specific JDBC programming topics. For links to WebLogic Server documentation and resources for this topic, see "Related Documentation" on page 1-2.

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.

Guide to this Document

- This chapter, Chapter 1, “Introduction and Roadmap,” introduces the organization of this guide and lists new features in the current release.
- Chapter 2, “Configuring WebLogic JDBC Resources,” which explains WebLogic JDBC configuration.
- Chapter 3, “Configuring JDBC Data Sources,” which describes WebLogic JDBC data source configuration.
- Chapter 4, “Configuring JDBC Multi Data Sources,” which describes WebLogic JDBC multi data source configuration.
- Chapter 5, “Using Third-Party JDBC Drivers with WebLogic Server,” which describes how to use JDBC driver from other sources in your WebLogic JDBC configuration.
- Chapter 6, “Monitoring WebLogic JDBC Resources,” which describes how to monitor JDBC resources, gather profile information about database connection usage, and enable JDBC debugging.
- Chapter 7, “Managing WebLogic JDBC Resources,” which describes how to administer data sources.
- Appendix A, “Configuring JDBC Application Modules for Deployment,” which describes how to package a WebLogic JDBC module with your enterprise application.
- Appendix B, “Using WebLogic Server with Oracle RAC,” which describes how to configure WebLogic Server for use with Oracle Real Application Clusters.

Related Documentation

This document contains JDBC-specific configuration and administration information.

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

- *Programming WebLogic JDBC* is a guide to JDBC API programming with WebLogic Server.
- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.

- *Deploying Applications to 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, BEA Systems provides a variety of JDBC code samples and tutorials that show JDBC configuration and API use, and provide practical instructions on how to perform key JDBC development tasks.

Avitek Medical Records Application (MedRec) and Tutorials

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 BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

JDBC Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

Additional JDBC Examples Available for Download

Additional API examples for download at `http://codesamples.projects.dev2dev.bea.com`. These examples are distributed as ZIP files that you can unzip into an existing WebLogic Server samples directory structure.

You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information at `https://codesample.projects.dev2dev.bea.com`.

New and Changed JDBC Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in release, see "What's New in WebLogic Server 10.0" in *Release Notes*.

CHAPTER 2

Configuring WebLogic JDBC Resources

The following sections describe WebLogic JDBC resources, how they are configured, and how those resources apply to a WebLogic domain:

- "Understanding JDBC Resources in WebLogic Server" on page 2-1
- "Ownership of Configured JDBC Resources" on page 2-3
- "JDBC Configuration Files" on page 2-3
- "JMX and WLST Access for JDBC Resources" on page 2-8
- "Overview of Clustered JDBC" on page 2-14

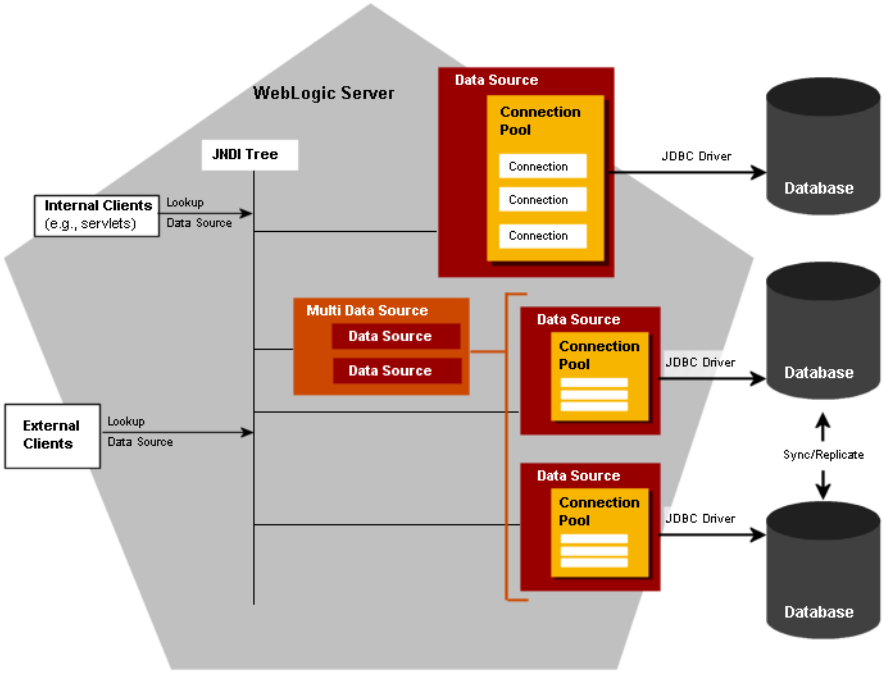
Understanding JDBC Resources in WebLogic Server

In WebLogic Server, you can configure database connectivity by configuring JDBC data sources and multi data sources and then targeting or deploying the JDBC resources to servers or clusters in your WebLogic domain.

Each data source that you configure contains a pool of database connections that are created when the data source instance is created—when it is deployed or targeted, or at server startup. Applications lookup a data source on the JNDI tree or in the local application context (`java:comp/env`), depending on how you configure and deploy the object, and then request a database connection. When finished with the connection, the application calls `connection.close()`, which returns the connection to the connection pool in the data source.

Figure 2-1 shows a data source and a multi data source targeted to a WebLogic Server instance.

Figure 2-1 JDBC Data Source Architecture



For more information about data sources in WebLogic Server, see "Configuring JDBC Data Sources" on page 3-1.

A multi data source is an abstraction around a data sources that provides load balancing or failover processing between the data sources associated with the multi data source. Multi data sources are bound to the JNDI tree or local application context just like data sources are bound to the JNDI tree. Applications lookup a multi data source on the JNDI tree or in the local application context (`java:comp/env`) just like they do for data sources, and then request a database connection. The multi data source determines which data source to use to satisfy the request depending on the algorithm selected in the multi data source configuration: load balancing or failover. For more information about multi data sources, see "Configuring JDBC Multi Data Sources" on page 4-1.

Ownership of Configured JDBC Resources

A key to understanding WebLogic JDBC configuration and management is that *who* creates a JDBC resource or *how* a JDBC resource is created determines how a resource is deployed and modified. Both WebLogic Administrators and programmers can create JDBC resources:

- WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JDBC modules. These JDBC modules are considered *system modules*. See "JDBC System Modules" on page 2-4 for more details.
- Programmers create modules in a development tool that supports creating an XML descriptor file, then package the JDBC modules with an application and pass the application to a WebLogic Administrator to deploy. These JDBC modules are considered *application modules*. See "JDBC Application Modules" on page 2-6 for more details.

Table 2-1 lists the JDBC module types and how they can be configured and modified.

Table 2-1 JDBC Module Types and Configuration and Management Options

Module Type	Created with	Add/Remove Modules with Administration Console	Modify with JMX (remotely)	Modify with JSR-88 (non-remotely)	Modify with Administration Console
System	Administration Console or WLST	Yes	Yes	No	Yes—via JMX
Application	BEA Workshop for WebLogic Platform, another IDE, or an XML editor	No	No	Yes—via a deployment plan	Yes—via a deployment plan

JDBC Configuration Files

WebLogic JDBC configuration is stored in XML documents that conform to the `weblogic-jdbc.xsd` schema (available at `http://www.bea.com/ns/weblogic/920/weblogic-jdbc.xsd`). You create and manage JDBC resources either as system modules or as application modules. JDBC application modules

are a WebLogic-specific extension of Java EE modules and can be configured either within a Java EE application or as stand-alone modules.

Regardless of whether you are using JDBC system modules or JDBC application modules, each JDBC data source or multi data source is represented by an XML file (a module).

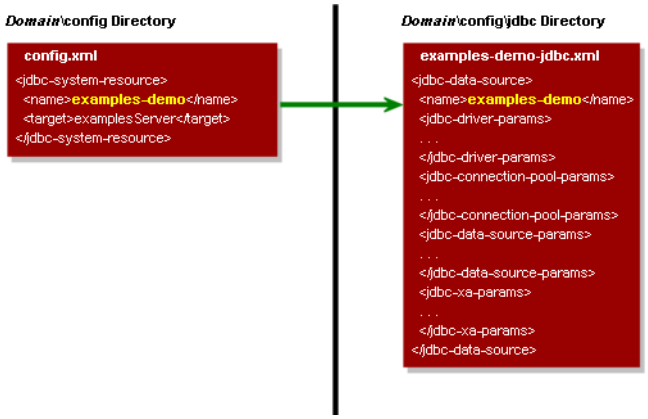
JDBC System Modules

When you create a JDBC resource (data source or multi data source) using the Administration Console or using the WebLogic Scripting Tool (WLST), WebLogic Server creates a JDBC module in the `config/jdbc` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file. The JDBC module conforms to the `weblogic-jdbc.xsd` schema (available at `http://www.bea.com/ns/weblogic/920/weblogic-jdbc.xsd`).

JDBC resources that you configure this way are considered *system modules*. System modules are owned by an Administrator, who can delete, modify, or add similar resources at any time. System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications deployed on the same targets and to client applications. System modules are also accessible through JMX as `JDBCSystemResourceMBeans`.

Data source system modules are included in the domain's `config.xml` file as a `JDBCSystemResource` element, which includes the name of the JDBC module file and the list of target servers and clusters on which the module is deployed. Figure 2-2 shows an example of a data source listing in a `config.xml` file and the module that it maps to.

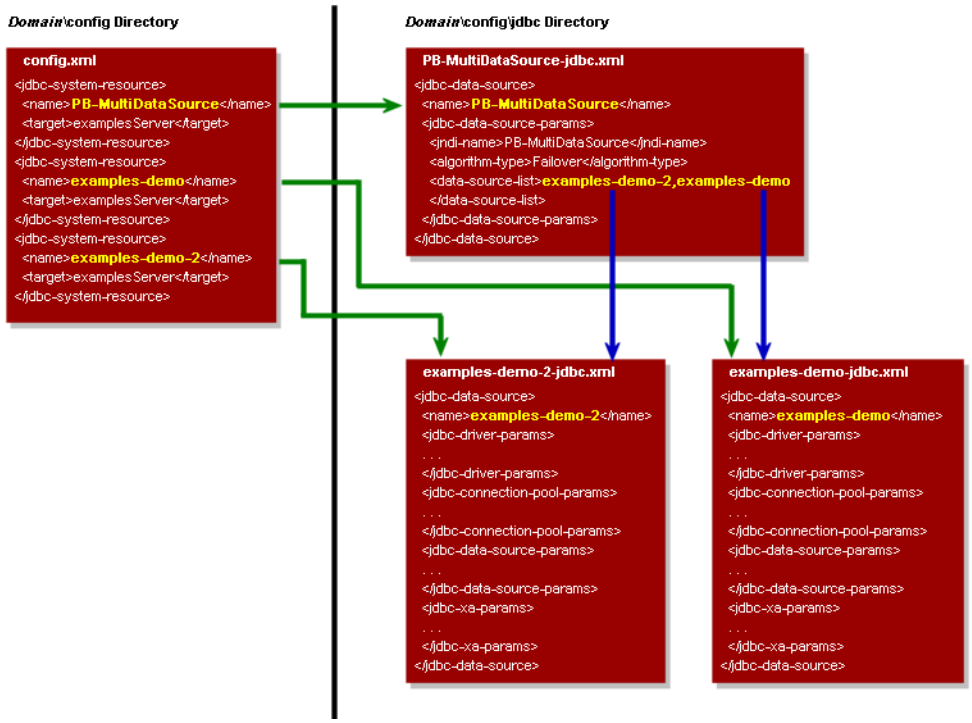
Figure 2-2 Reference from config.xml to a Data Source System Module



In this illustration, the `config.xml` file lists the `examples-demo` data source as a `jdbc-system-resource` element, which maps to the `examples-demo.xml` module in the *`domain`*`\config\jdbc` folder.

Similarly, multi data source system modules are included in the domain's `config.xml` file as a `jdbc-system-resource` element. The multi data source module includes a `data-source-list` parameter that maps to the data source modules used by the multi data source. The individual data source modules are also included in the `config.xml`. Figure 2-3 shows the relationship between elements in the `config.xml` file and the system modules in the `config/jdbc` directory.

Figure 2-3 Reference from config.xml to Multi Data Source and Data Source System Modules



In this illustration, the `config.xml` file lists three JDBC modules—one multi data source and the two data sources used by the multi data source, which are also listed within the multi data source module. Your application can look up any of these modules on the JNDI tree and request a

database connection. If you look up the multi data source, the multi data source determines which of the other data sources to use to supply the database connection, depending on the data sources in the `data-source-list` parameter, the order in which the data sources are listed, and the algorithm specified in the `algorithm-type` parameter. For more information about multi data sources, see "Configuring JDBC Multi Data Sources" on page 4-1.

JDBC Application Modules

JDBC resources can also be managed as application modules, similar to standard Java EE modules. A JDBC application module is simply an XML file that conforms to the `weblogic-jdbc.xsd` schema and represents a data source or a multi data source.

JDBC modules can be included as part of an Enterprise Application as a *packaged module*. Packaged modules are bundled with an EAR or exploded EAR directory, and are referenced in all appropriate deployment descriptors, such as the `weblogic-application.xml` and `ejb-jar.xml` deployment descriptors. The JDBC module is deployed along with the enterprise application, and can be configured to be available only to the enclosing application or to all applications. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments. With packaged JDBC modules, you can migrate your application and the required JDBC configuration from environment to environment, such as from a testing environment to a production environment, without opening an EAR file and without extensive manual JDBC reconfiguration.

In contrast to system resource modules, JDBC modules that are packaged with an application are owned by the developer who created and packaged the module, rather than the Administrator who deploys the module. This means that the Administrator has more limited control over packaged modules. When deploying a resource module, an Administrator can change resource properties that were specified in the module, but the Administrator cannot add or delete modules. (As with other Java EE modules, deployment configuration changes for a resource module are stored in a deployment plan for the module, leaving the original module untouched.)

By definition, packaged JDBC modules are included in an enterprise application, and therefore are deployed when you deploy the enterprise application. For more information about deploying applications with packaged JDBC modules, see *Deploying Applications to WebLogic Server*.

A JDBC application module can also be deployed as a stand-alone resource using the `weblogic.Deployer` utility or the Administration Console, in which case the resource is typically available to the server or cluster targeted during the deployment process. JDBC resources deployed in this manner are called *stand-along modules* and can be reconfigured using

the Administration Console or a JSR-88 compliant tool, but are unavailable through JMX or WLST.

Stand-alone JDBC modules promote sharing and portability of JDBC resources. You can create a data source configuration and distribute it to other developers. Stand-alone JDBC modules can also be used to move JDBC configuration between domains, such as between the development domain and the staging domain.

For more information about JDBC application modules, see "Configuring JDBC Application Modules for Deployment" on page A-1.

For information about deploying stand-alone JDBC modules, see "Deploying JDBC, JMS, WLDF Application Modules."

JDBC Module File Naming Requirements

All WebLogic JDBC module files must end with the -jdbc.xml suffix, such as `examples-demo-jdbc.xml`. WebLogic Server checks the file name when you deploy the module. If the file does not end in `-jdbc.xml`, the deployment will fail and the server will not boot.

JDBC Modules in Versioned Applications

When you use production redeployment (versioning) to deploy a version of an application that includes a packaged JDBC module, WebLogic Server identifies the data source defined in the JDBC module with a name in the following format:

```
application_id#version_id@module_name@data_source_name
```

This name is used for data source runtime MBeans and for registering the data source instance with the WebLogic Server transaction manager.

If transactions in a retiring version of an application time out and the version of the application is then undeployed, you may have to manually resolve any pending or incomplete transactions on the data source in the retired version of the application. After a data source is undeployed (in this case, with the retired version of the application), the WebLogic Server transaction manager cannot recover pending or incomplete transactions.

For more information about production redeployment, see:

- "Developing Applications for Production Redeployment" in *Developing Applications with WebLogic Server*

- "Using Production Redeployment to Update Applications" in *Deploying Applications to WebLogic Server*

JDBC Schema

In support of the modular deployment model for JDBC resources in WebLogic Server, BEA provides a schema for WebLogic JDBC objects: `weblogic-jdbc.xsd`. When you create JDBC resource modules (descriptors), the modules must conform to the schema. IDEs and other tools can validate JDBC resource modules based on the schema.

The schema is available at `http://www.bea.com/ns/weblogic/920/weblogic-jdbc.xsd`.

JMX and WLST Access for JDBC Resources

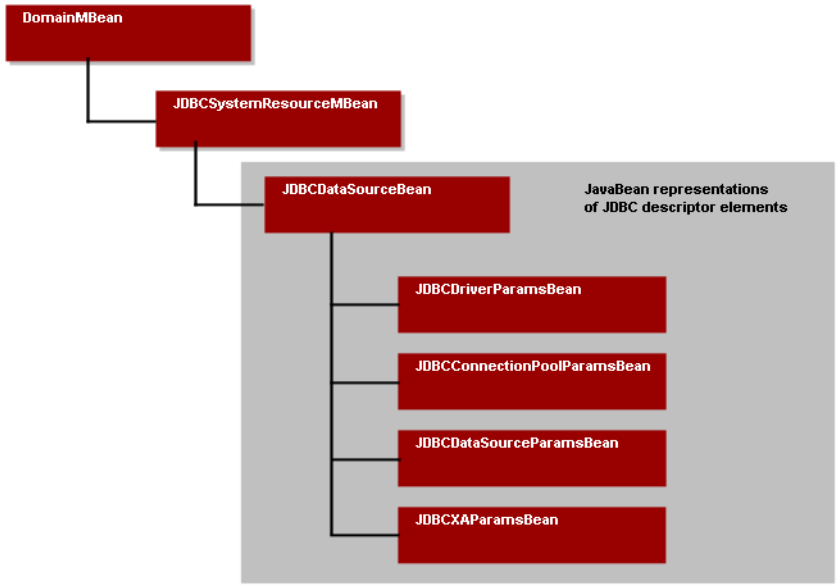
When you create JDBC resources using the Administration Console or WLST, WebLogic Server creates MBeans (Managed Beans) for each of the resources. You can then access these MBeans using JMX or the WebLogic Scripting Tool (WLST). See *Developing Custom Management Utilities with JMX* and *WebLogic Scripting Tool* for more information.

- "JDBC MBeans for System Resources" on page 2-8
- "JDBC Management Objects in the Java EE Management Model (JSR-77 Support)" on page 2-9
- "Using WLST to Create JDBC System Resources" on page 2-10
- "How to Modify and Monitor JDBC Resources" on page 2-13
- "Best Practices when Using WLST to Configure JDBC" on page 2-13

JDBC MBeans for System Resources

Figure 2-4 shows the hierarchy of the MBeans for JDBC objects in a WebLogic domain.

Figure 2-4 JDBC Bean Tree



The `JDBCSystemResourceMBean` is a container for the JavaBeans created from a data source module. However, all JMX access for a JDBC data source is through the `JDBCSystemResourceMBean`. You cannot directly access the individual JavaBeans created from the data source module.

JDBC Management Objects in the Java EE Management Model (JSR-77 Support)

WebLogic Server JDBC supports JSR-77, which defines the Java EE Management Model. The Java EE Management Model is used for monitoring the runtime state of a Java EE Web application server and its resources. You can access the Java EE Management Model to monitor resources, including the WebLogic JDBC system as a whole, JDBC drivers loaded into memory, and JDBC data sources.

To comply with the specification, BEA added the following runtime MBean types for WebLogic JDBC:

- `JDBCServiceRuntimeMBean`—Which represents the JDBC subsystem and provides methods to access the list of `JDBCDriverRuntimeMBeans` and `JDBCDataSourceRuntimeMBeans` currently available in the system.
- `JDBCDriverRuntimeMBean`—Which represents a JDBC driver that the server loaded into memory.
- `JDBCDataSourceRuntimeMBeans`—Which represents a JDBC data source deployed on a server or cluster.

Note: WebLogic JDBC runtime MBeans do *not* implement the optional Statistics Provider interfaces specified by JSR-77.

For more information about using the Java EE management model with WebLogic Server, see *Monitoring and Managing with the Java EE Management APIs*.

Using WLST to Create JDBC System Resources

Basic tasks you need to perform when creating JDBC resources with the WLST are:

- Start an edit session.
- Create a JDBC system module that includes JDBC system resources, such as pools, datasources, multi datasources, and JDBC drivers.
- Target your JDBC system module.

Listing 2-1 WLST Script to Create JDBC Resources

```
#-------------------------------------------------------------------
# Create JDBC
# The prefix specifies the prefix on property names.
#  Example: for property "mypool.Name=mypool", the prefix would be "mypool."
#-------------------------------------------------------------------

import sys
from java.lang import System

print "@@@ Starting the script ..."
global props
```

```
url = sys.argv[1]
usr = sys.argv[2]
password = sys.argv[3]

connect(usr,password, url)
edit()
startEdit()

servermb=getMBean("Servers/examplesServer")
   if servermb is None:
      print '@@@ No server MBean found'
else:
   def addJDBC(prefix):

   print("")
          print("*** Creating JDBC with property prefix " + prefix)

       # Create the Connection Pool.  The system resource will have
       # generated name of <PoolName>+"-jdbc"

          myResourceName = props.getProperty(prefix+"PoolName")
          print("Here is the Resource Name: " + myResourceName)

          jdbcSystemResource =
wl.create(myResourceName,"JDBCSystemResource")
       myFile = jdbcSystemResource.getDescriptorFileName()
          print ("HERE IS THE JDBC FILE NAME: " + myFile)

       jdbcResource = jdbcSystemResource.getJDBCResource()
          jdbcResource.setName(props.getProperty(prefix+"PoolName"))

       # Create the DataSource Params
          dpBean = jdbcResource.getJDBCDataSourceParams()
          myName=props.getProperty(prefix+"JNDIName")
   dpBean.setJNDINames([myName])

       # Create the Driver Params
```

```
            drBean = jdbcResource.getJDBCDriverParams()
            drBean.setPassword(props.getProperty(prefix+"Password"))
            drBean.setUrl(props.getProperty(prefix+"URLName"))
            drBean.setDriverName(props.getProperty(prefix+"DriverName"))

            propBean = drBean.getProperties()
            driverProps = Properties()

   driverProps.setProperty("user",props.getProperty(prefix+"UserName"))

            e = driverProps.propertyNames()
            while e.hasMoreElements() :
                      propName = e.nextElement()
                      myBean = propBean.createProperty(propName)
                      myBean.setValue(driverProps.getProperty(propName))

        # Create the ConnectionPool Params
            ppBean = jdbcResource.getJDBCConnectionPoolParams()

   ppBean.setInitialCapacity(int(props.getProperty(prefix+"InitialCapacity
"")))

   ppBean.setMaxCapacity(int(props.getProperty(prefix+"MaxCapacity")))

   ppBean.setCapacityIncrement(int(props.getProperty(prefix+"CapacityIncre
ment")))

            if not props.getProperty(prefix+"ShrinkPeriodMinutes") == None:
      ppBean.setShrinkFrequencySeconds(int(props.getProperty(prefix+"Shrin
kPeriodMinutes")))
                        if not props.getProperty(prefix+"TestTableName")
== None:

      ppBean.setTestTableName(props.getProperty(prefix+"TestTableName"))

            if not props.getProperty(prefix+"LoginDelaySeconds") == None:
      ppBean.setLoginDelaySeconds(int(props.getProperty(prefix+"LoginDelay
Seconds")))
```

```
       # Adding KeepXaConnTillTxComplete to help with in-doubt
transactions.
          xaParams = jdbcResource.getJDBCXAParams()
          xaParams.setKeepXaConnTillTxComplete(1)

       # Add Target

   jdbcSystemResource.addTarget(wl.getMBean("/Servers/examplesServer"))
.
.
.
```

How to Modify and Monitor JDBC Resources

You can modify or monitor JDBC objects and attributes by using the appropriate method available from the MBean.

- You can modify JDBC objects and attributes using the set, target, untarget, and delete methods.
- You can monitor JDBC runtime objects using get methods.

For more information, see Navigating and Editing MBeans in the *WebLogic Scripting Tool.*

Best Practices when Using WLST to Configure JDBC

This section provides best practices information when using WLST to configure JDBC resources:

- Trap for Null MBean objects (such as pools, datasources, drivers) before trying to manipulate the MBean object.
- BEA provides sample scripts and utilities to configure WebLogic domain resources using WLST Offline and/or WLST Online. For more information, see the *wlst Project Home* at https://wlst.projects.dev2dev.bea.com/.

Overview of Clustered JDBC

You can target or deploy JDBC resources to a cluster to improve the availability of cluster-hosted applications. For information about JDBC objects in a clustered environment, see "JDBC Connections" in *Using WebLogic Server Clusters*.

Multi data sources are supported for use in clusters. However, note that multi data sources can only use data sources in the same JVM. Multi data sources cannot use data sources from other cluster members.

CHAPTER 3

Configuring JDBC Data Sources

This section includes the following information:

- "Understanding JDBC Data Sources" on page 3-1
- "Creating a JDBC Data Source" on page 3-2
- "Transaction Options" on page 3-4
- "Connection Pool Features" on page 3-13
- "Setting Database Security Credentials" on page 3-15
- "Tuning Data Source Connection Pool Options" on page 3-19
- "Deploying Data Sources on Servers and Clusters" on page 3-33
- "Minimizing Server Startup Hang Caused By an Unresponsive Database" on page 3-33
- "Security for JDBC Data Sources" on page 3-34
- "JDBC Data Source Factories (Deprecated)" on page 3-35

Understanding JDBC Data Sources

In WebLogic Server, you configure database connectivity by adding data sources to your WebLogic domain. WebLogic JDBC data sources provide database access and database connection management. Each data source contains a pool of database connections that are created when the data source is created and at server startup. Applications reserve a database

connection from the data source by looking up the data source on the JNDI tree or in the local application context and then calling `getConnection()`. When finished with the connection, the application should call `connection.close()` as early as possible, which returns the database connection to the pool for other applications to use.

Data sources and their connection pools provide connection management processes that help keep your system running and performant.You can set options in the data source to suit your applications and your environment. The following sections describe these options and how to enable them.

Creating a JDBC Data Source

To create a JDBC data source in your WebLogic domain, you can use the Administration Console or the WebLogic Scripting Tool (WLST). See the following for more information:

- "Create JDBC data sources" in the *Administration Console Online Help*
- The sample WLST script *SAMPLES_HOME*`\server\examples\src\examples\wlst\online\jdbc_data_source_creation.py`, where *SAMPLES_HOME* refers to the main examples directory of your WebLogic Server installation. See "WLST Online Sample Scripts" in *WebLogic Scripting Tool*

Note: WLST replaced the weblogic.Admin command line utility. The WebLogic Server examples that are optionally installed with WebLogic Server contain sample scripts that can be used in place of the weblogic.Admin JDBC commands. If installed, the example scripts are available at *WL_HOME*`\samples\server\examples\src\examples\wlst\online`, where *WL_HOME* refers to the main WebLogic directory, such as `C:\bea\wlserver_10.0`.

For more information about JDBC data source attributes, see the "JDBCDataSourceBean" and all of its child MBeans in the *WebLogic Server MBean Reference*

- JDBC data source reference pages in the *Administration Console Online Help*:
  - JDBC Data Source: Configuration: General
  - JDBC Data Source: Configuration: Connection Pool
  - JDBC Data Source: Configuration: Transaction
  - JDBC Data Source: Configuration: Diagnostics
  - JDBC Data Source: Configuration: Identity Options

- JDBC Data Sources: Targets
- JDBC Data Source: Security: Roles
- JDBC Data Source: Security: Policies
- JDBC Data Sources: Security: Credential Mapping

Notes: JDBC drivers listed in the Create JDBC Data Source pages in the Administration Console are not necessarily certified for use with WebLogic Server. In keeping with the goal of the Create JDBC Data Source pages, JDBC drivers are listed as a convenience to help you create connections to many of the database management systems available.

You must install JDBC drivers in order to use them to create database connections in a data source on each server on which the data source is deployed. Drivers are listed in the Create JDBC Data Source pages in the Administration Console with known required configuration options to help you configure a data source. The JDBC drivers in the list are not necessarily installed. Driver installation can include setting system Path, Classpath, and other environment variables. See "Setting the Environment for a Type-4 Third-Party JDBC Driver" on page 5-2.

When a JDBC driver is updated, configuration requirements may change. The Create JDBC Data Source pages in the Administration Console use known configuration requirements at the time the WebLogic Server software was released. If configuration options for your JDBC driver have changed, you may need to manually override the configuration options when creating the data source or in the property pages for the data source after it is created.

General Data Source Options

JDBC data sources include options that determine the identity of the data source, way the data is handled on a database connection, and the way transactions are handled when a connection from the data source is used in a global transaction. You can view general options for a JDBC data source on the JDBC Data Source: Configuration: General page in the Administration Console. You can also access these options from the JDBCDataSourceParamsBean, which is a child of the JDBCDataSourceBean.

Selecting a JDBC Driver

When deciding which JDBC driver to use to connect to a database, you should try drivers from various vendors in your environment. In general, JDBC driver performance is dependent on many factors, especially the SQL code used in applications and the JDBC driver implementation.

For information about supported JDBC drivers, see "Supported Database Configurations" in *Supported Configurations for WebLogic Platform 10.0*.

JDBC Data Source Names

JDBC data source names are used to identify the data source within the WebLogic domain. For system resource data sources, names must be unique among all other JDBC system resources, including data sources and multi data sources. To avoid naming conflicts, data source names should also be unique among other configuration object names, such as servers, clusters, and JMS queues, topics, and servers. For JDBC application modules scoped to an application, data source names must be unique among JDBC data sources and multi data sources that are similarly scoped.

Binding a Data Source to the JNDI Tree with Multiple Names

In WebLogic Server 9.0 and later releases, you can configure a data source so that it binds to the JNDI tree with multiple names. You can use a multi-JNDI-named data source in place of legacy configurations that included multiple data sources that pointed to a single JDBC connection pool.

To add JNDI names to an existing data source using the Administration Console, add names to the JNDI Name attribute with each JNDI name on a separate line. You must either restart the system after making your change or undeploy the data source before making the change, and then redeploy after making the change. Follow the instructions below.

1. On the JDBC Data Source → Configuration → General page in the Administration Console, in JNDI Name, enter the names you want to use to bind the data source to the JDNI tree with each name on a separate line. For example:

   ```
   name1
   name2
   name3
   ```

2. Click Save.

After you activate your changes, you will need to redeploy the data source or restart your server before the changes will take effect.

Transaction Options

When you configure a JDBC data source using the Administration Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver:

- **For XA drivers**, the system automatically selects the **Two-Phase Commit** protocol for global transaction processing.
- **For non-XA drivers**, local transactions are supported by definition, and WebLogic Server offers the following options

  Supports Global Transactions: (selected by default) Select this option if you want to use connections from the data source in global transactions, even though you have not selected an XA driver. See "Enabling Support for Global Transactions with a Non-XA JDBC Driver" on page 3-5 for more information.

  When you select Supports Global Transactions, you must also select the protocol for WebLogic Server to use for the transaction branch when processing a global transaction:

  – **Logging Last Resource**: With this option, the transaction branch in which the connection is used is processed as the last resource in the transaction and is processed as a local transaction. Commit records for two-phase commit (2PC) transactions are inserted in a table on the resource itself, and the result determines the success or failure of the prepare phase of the global transaction. This option offers some performance benefits and greater data safety than Emulate Two-Phase Commit, but it has some limitations. See "Understanding the Logging Last Resource Transaction Option" on page 3-6.

    Note: Logging Last Resource is *not* supported for data sources used by a multi data source.

  – **Emulate Two-Phase Commit**: With this option, the transaction branch in which the connection is used always returns success for the prepare phase of the transaction. It offers performance benefits, but also has risks to data in some failure conditions. Select this option only if your application can tolerate heuristic conditions. See "Understanding the Emulate Two-Phase Commit Transaction Option" on page 3-11.

  – **One-Phase Commit**: (selected by default) With this option, a connection from the data source can be the only participant in the global transaction and the transaction is completed using a one-phase commit optimization. If more than one resource participates in the transaction, an exception is thrown when the transaction manager calls `XAResource.prepare` on the 1PC resource.

Enabling Support for Global Transactions with a Non-XA JDBC Driver

If you use global transactions in your applications, you should use an XA JDBC driver to create database connections in the JDBC data source. If an XA driver is unavailable for your database,

or you prefer not to use an XA driver, you should enable support for global transactions in the data source. You should also enable support for global transaction if your applications meet any of the following criteria:

- Use the EJB container in WebLogic Server to manage transactions
- Include multiple database updates within a single transaction
- Access multiple resources, such as a database and the Java Messaging Service (JMS), during a transaction
- Use the same data source on multiple servers (clustered or non-clustered)

With an EJB architecture, it is common for multiple EJBs that are doing database work to be invoked as part of a single transaction. Without XA, the only way for this to work is if all transaction participants use the exact same database connection. When you enable global transactions and select either Logging Last Resource or Emulate Two-Phase Commit, WebLogic Server internally uses the JTS driver to make sure all EJBs use the same database connection within the same transaction context without requiring you to explicitly pass the connection from EJB to EJB.

If multiple EJBs are participating in a transaction and you do not use an XA JDBC driver for database connections, configure a Data Source with the following options:

- Supports Global Transactions selected
- Logging Last Resource or Emulate Two-Phase Commit selected

This configuration will force the JTS driver to internally use the same database connection for all database work within the same transaction.

With XA (requires an XA driver), EJBs can use a different database connection for each part of the transaction. WebLogic Server coordinates the transaction using the two-phase commit protocol, which guarantees that all or none of the transaction will be completed.

Understanding the Logging Last Resource Transaction Option

WebLogic Server supports the Logging Last Resource (LLR) transaction optimization through JDBC data sources. LLR is a performance enhancement option that enables one non-XA resource to participate in a global transaction with the same ACID guarantee as XA. LLR is a refinement of the “Last Agent Optimization.” It differs from Last Agent Optimization in that it is transactionally safe. The LLR resource uses a local transaction for its transaction work. The

WebLogic Server transaction manager prepares all other resources in the transaction and then determines the commit decision for the global transaction based on the outcome of the LLR resource's local transaction.

The LLR optimization improves performance by:

- Removing the need for an XA JDBC driver to connect to the database. XA JDBC drivers are typically inefficient compared to non-XA JDBC drivers.
- Reducing the number of processing steps to complete the transaction, which also reduces network traffic and the number of disk I/Os.
- Removing the need for XA processing at the database level

When a connection from a data source configured for LLR participates in a two-phase commit (2PC) global transaction, the WebLogic Server transaction manager completes the transaction by:

- Calling prepare on all other (XA-compliant) transaction participants.
- Inserting a commit record to a table on the LLR participant (rather than to the file-based transaction log).
- Committing the LLR participant's local transaction (which includes both the transaction commit record insert and the application's SQL work).
- Calling commit on all other transaction participants.

For a one-phase commit (1PC) global transaction, LLR eliminates the XA overhead by using a local transaction to complete the database operations, but no 2PC transaction record is written to the database.

The Logging Last Resource optimization maintains data integrity by writing the commit record on the LLR participant. If the transaction fails during the local transaction commit, the WebLogic Server transaction manager rolls back the transaction on all other transaction participants. For failure recovery, the WebLogic Server transaction manager reads the transaction log on the LLR resource along with other transaction log files in the default store and completes any transaction processing as necessary. Work associated with XA participants is committed if a commit record exists, otherwise their work is rolled back.

For instructions on how to create an LLR-enabled JDBC data source, see "Create LLR-enabled JDBC data sources" in the *Administration Console Online Help*. For more details about the Logging Last Resource transaction processing, see "Logging Last Resource Transaction Optimization" in *Programming WebLogic JTA*.

Advantages to Using the Logging Last Resource Optimization

Depending on your environment, you may want to consider the LLR transaction protocol in place of the two-phase commit protocol for transaction processing because of its performance benefits. The LLR transaction protocol offers the following advantages:

- Allows non-XA JDBC drivers and even non-XA–capable databases to safely participate in two-phase commit transactions.
- Eliminates the database's use of the XA protocol.
- Performs better than JDBC XA connections.
- Reduces the length of time that database row locks are held.
- Always commits database work prior to other XA work. In XA transactions, these operations are committed in parallel, so, for example, when a JMS send participates in the transaction, the JMS message may be delivered before database work commits. With LLR, the database work in the local transaction is completed before all other transaction work.
- Has no increased risk of heuristic hazards, unlike the Emulate Two-Phase Commit option for a JDBC data source.

Note: The LLR optimization provides a significant increase in performance for insert, update, and delete operations. However, for read operations with LLR, performance is somewhat slower than read operations with XA. For best performance, you may want to configure a non-LLR JDBC data source for read-only operations.

For more information about performance tuning with LLR, see "Optimizing Performance with LLR" in *Programming WebLogic JTA*.

Enabling the Logging Last Resource Transaction Optimization

To enable the LLR transaction optimization, you create a JDBC data source with the Logging Last Resource transaction protocol, then use database connections from the data source in your applications. WebLogic Server automatically creates the required table on the database.

See "Create LLR-enabled JDBC data sources" in the *Administration Console Online Help*.

Programming Considerations and Limitations for LLR Data Sources

You use JDBC connections from an LLR-enabled data source in an application as you would use JDBC connections from any other data source: *after* beginning a transaction, you look up the data source on the JNDI tree, then request a connection from the data source. However, with the LLR optimization, WebLogic Server internally manages the connection request and handles the

transaction processing differently than in an XA transaction. For more information about how Logging Last Resource works, see "Logging Last Resource Transaction Optimization" in *Programming WebLogic JTA*.

Note the following:

- When programming with an LLR data source, you must start the global transaction before calling getConnection on the LLR data source. If you call getConnection before starting the global transaction, all operations on the connection will be made outside of the global transaction.
- Only one internal JDBC LLR connection is reserved per transaction. And that connection is used throughout the transaction processing.
- The reserved connection is always hosted on the transaction's coordinator server. Make sure that the data source is targeted to the coordinating server or to the cluster. Also see "Optimizing Performance with LLR" in *Programming WebLogic JTA*.
- For additional JDBC connection requests within the transaction from a same-named data source, operations are routed to the reserved connection from the original connection request, even if the subsequent connection request is made on a different instance of the data source (i.e., a data source deployed on a different server than the original data source that supplied the connection for the first request). Note the following:
  - Routed LLR connections may be less capable and less performant than locally hosted XA connections. (See "Possible Performance Loss with Non-XA Resources in Multi-Server Configurations" on page 3-13.)
  - Connection request routing limits the number of concurrent transactions. The maximum number of concurrent LLR transactions is equal to the configured size (MaxCapacity) of the coordinator's JDBC LLR data source.
  - Routed connections have less capability than local connections, and may fail as a result. Specifically, non-serializable "custom" data types within a query ResultSet may fail.
- Only instances of a single LLR data source may participate in a particular transaction. A single LLR data source may have instances on multiple WebLogic servers, and two data sources are considered to be the same if they have the same configured name. If more than one LLR data source instance is detected and they are not instances of the same data source, the transaction manager will roll back the transaction.
- Resource adapters (connectors) that implement the `weblogic.transaction.nonxa.NonXAResource` interface cannot participate in global transaction in which an LLR resource also participates because both must be the last

resource in the transaction. If both resource types participate in the same transaction, the transaction `commit()` method throws a `javax.transaction.RollbackException` when this conflict is detected.

- Because the LLR connection uses a separate *local* transaction for database processing, any changes made (and locks held) to the same database using an XA connection are not visible during the LLR processing even though all of the processing occurs in the same *global* transaction. In some cases, this can cause deadlocks in the database. You should not combine XA and LLR processing in the same database in a single global transaction.
- Connections from an LLR data source cannot participate in transactions coordinated by foreign transaction managers, such as a transaction started by a remote object request broker or by Tuxedo.
- Global transactions cannot span to another legacy domain that includes a data source with the same name as an LLR data source.
- For JDBC LLR 2PC transactions, if the transaction data is too large to fit in the LLR table, the transaction will fail with a rollback exception thrown during commit. This can occur if your application adds many transaction properties during transaction processing. (See "BEA WebLogic Extensions to JTA.") Your database administrator can manually create a table with larger columns if this occurs.

Administrative Considerations and Limitations for LLR Data Sources

Consider the following requirements and limitations when configuring an LLR-enabled JDBC data source. For more information about how Logging Last Resource works, see "Logging Last Resource Transaction Optimization" in *Programming WebLogic JTA*.

- There is one LLR table per server:
  - Multiple LLR data sources may share a table.
  - WebLogic Server automatically creates the table if it is not found.
  - Default name is `WL_LLR_SERVERNAME`. You can configure the table name in the Administration Console on the Server > Configuration > General tab under Advanced options.
- A server **will not boot** if the database is down or the LLR table is unreachable during boot.
- Multiple servers must *not* share the same LLR table. Boot checks to ensure domain and server name match the domain and server name stored in the table when the table is created. If WebLogic Server detects that more than one server is sharing the same LLR table, WebLogic Server will shut down one or more of the servers.

- LLR supports server migration and transaction recovery service migration. To use the transaction recovery service migration, ensure that each LLR resource be targeted to either the cluster or the set of candidate servers in the cluster. Recovering Transactions for a Failed Clustered Server in "Transaction Recovery After a Server Fails" in *Programming WebLogic JTA*.
- The LLR transaction option is not permitted for use in JDBC application modules.
- The LLR transaction option is not supported for use in data sources used by a multi data source.
- If you use credential mapping on an LLR data source, all mapped users must have write permissions on the LLR table.
- You cannot use a JDBC XA driver to create database connections in a JDBC LLR data source. If the JDBC driver used in a JDBC LLR data source supports XA, a warning message is logged, and the data source participates in transactions as a full XA resource rather than as an LLR resource.
- Transaction statistics for LLR resources are tracked under "NonXAResource." See "View transaction statistics for non-XA resources" in the *Administration Console Online Help*.

Understanding the Emulate Two-Phase Commit Transaction Option

If you need to support distributed transactions with a JDBC data source, but there is no available XA-compliant driver for your DBMS, you can select the Emulate Two-Phase Commit for non-XA Driver option for a data source to emulate two-phase commit for the transactions in which connections from the data source participate. This option is an advanced option on the JDBC Data Source → Configuration → General tab.

When the Emulate Two-Phase Commit for non-XA Driver option is selected (`EnableTwoPhaseCommit` is set to `true`), the non-XA JDBC resource always returns `XA_OK` during the `XAResource.prepare()` method call. The resource attempts to commit or roll back its local transaction in response to subsequent `XAResource.commit()` or `XAResource.rollback()` calls. If the resource commit or rollback fails, a heuristic error results. Application data may be left in an inconsistent state as a result of a heuristic failure.

When the Emulate Two-Phase Commit for non-XA Driver option is not selected in the Console (`EnableTwoPhaseCommit` is set to `false`), the non-XA JDBC resource causes `XAResource.prepare()` to fail. When there is only one resource participating in a transaction,

the one phase optimization bypasses `XAResource.prepare()`, and the transaction commits successfully in most instances.

Note: There are risks to data integrity when using the Emulate Two-Phase Commit for non-XA Driver option. BEA recommends that you use an XA-compliant JDBC driver or the Logging Last Resource option rather than use the Emulate Two-Phase Commit option. Make sure you consider the risks below before enabling this option.

This non-XA JDBC driver support is often referred to as the "JTS driver" because WebLogic Server uses the WebLogic JTS Driver internally to support the feature. For more information about the WebLogic JTS Driver, see "Using the WebLogic JTS Driver" in *Programming WebLogic JDBC*.

Limitations and Risks When Emulating Two-Phase Commit Using a Non-XA Driver

WebLogic Server supports the participation of non-XA JDBC resources in global transactions with the Emulate Two-Phase Commit data source transaction option, but there are limitations that you must consider when designing applications to use such resources. Because a non-XA driver does not adhere to the XA/2PC contracts and only supports one-phase commit and rollback operations, WebLogic Server (through the JTS driver) has to make compromises to allow the resource to participate in a transaction controlled by the Transaction Manager.

Consider the following limitations and risks before using the Emulate Two-Phase Commit for non-XA Driver option.

Heuristic Completions and Data Inconsistency

When Emulate Two-Phase Commit is selected for a non-XA resource, (`enableTwoPhaseCommit = true`), the prepare phase of the transaction for the non-XA resource always succeeds. Therefore, the non-XA resource does not truly participate in the two-phase commit (2PC) protocol and is susceptible to failures. If a failure occurs in the non-XA resource after the prepare phase, the non-XA resource is likely to roll back the transaction while XA transaction participants will commit the transaction, resulting in a heuristic completion and data inconsistencies.

Because of the data integrity risks, the Emulate Two-Phase Commit option should only be used in applications that can tolerate heuristic conditions.

Cannot Recover Pending Transactions

Because a non-XA driver manipulates local database transactions only, there is no concept of a transaction pending state in the database with regard to an external transaction manager. When `XAResource.recover()` is called on the non-XA resource, it always returns an empty set of Xids (transaction IDs), even though there may be transactions that need to be committed or rolled back. Therefore, applications that use a non-XA resource in a global transaction cannot recover from a system failure and maintain data integrity.

Possible Performance Loss with Non-XA Resources in Multi-Server Configurations

Because WebLogic Server relies on the database local transaction associated with a particular JDBC connection to support non-XA resource participation in a global transaction, when the same JDBC data source is accessed by an application with a global transaction context on multiple WebLogic Server instances, the JTS driver will always route JDBC operations to the first connection established by the application in the transaction. For example, if an application starts a transaction on one server, accesses a non-XA JDBC resource, then makes a remote method invocation (RMI) call to another server and accesses a data source that uses the same underlying JDBC driver, the JTS driver recognizes that the resource has a connection associated with the transaction on another server and sets up an RMI redirection to the actual connection on the first server. All operations on the connection are made on the one connection that was established on the first server. This behavior can result in a performance loss due to the overhead associated with setting up these remote connections and making the RMI calls to the one physical connection.

Only One Non-XA Participant

When a non-XA resource (with Emulate Two-Phase Commit selected) is registered with the WebLogic Server Transaction Manager, it is registered with the name of the class that implements the XAResource interface. Since all non-XA resources with Emulate Two-Phase Commit selected use the JTS driver for the XAResource interface, all non-XA resources (with Emulate Two-Phase Commit selected) that participate in a global transaction are registered with the same name. If you use more than one non-XA resource in a global transaction, you will see naming conflicts or possible heuristic failures.

Connection Pool Features

Each JDBC data source has a pool of JDBC connections that are created when the data source is deployed or at server startup. Applications use a connection from the pool then return it when

finished using the connection. Connection pooling enhances performance by eliminating the costly task of creating database connections for the application.

The following sections include information about connection pool options for a JDBC data source.

- "Selecting a JDBC Driver" on page 3-14
- "Enabling JDBC Driver-Level Features" on page 3-15
- "Initializing Database Connections with SQL Code" on page 3-15

You can see more information and set these and other related options through the:

- JDBC Data Source: Configuration: Connection Pool page in the Administration Console
- JDBCConnectionPoolParamsBean, which is a child MBean of the JDBCDataSourceBean

Selecting a JDBC Driver

When creating a JDBC data source using the Administration Console, you are prompted to select a JDBC driver. The Administration Console provides the driver class name and helps you construct the URL as required by the driver. The driver you select must be in the classpath on all servers on which you intend to deploy the data source. Some but not all JDBC drivers listed in the Administration Console are shipped with WebLogic Server:

- Third-party JDBC drivers (see "Using Third-Party JDBC Drivers with WebLogic Server" on page 5-1):
  - Oracle Thin Driver (XA and non-XA)
  - Sybase jConnect
  - PointBase
  - MySQL (non-XA)
- WebLogic Type 4 JDBC Drivers from DataDirect for the following database management systems (see *WebLogic Type 4 JDBC Drivers*):
  - DB2
  - Informix
  - Microsoft SQL Server
  - Oracle

– Sybase

All of these drivers are referenced by the `weblogic.jar` manifest file and do not need to be explicitly defined in a server's classpath.

Enabling JDBC Driver-Level Features

WebLogic JDBC data sources support the `javax.sql.ConnectionPoolDataSource` interface implemented by JDBC drivers. You can enable driver-level features adding the property and its value to the Properties attribute in a JDBC data source. Driver-level properties in the Properties attribute are set on the driver's ConnectionPoolDataSource object.

Initializing Database Connections with SQL Code

When WebLogic Server creates database connections in a data source, the server can automatically run SQL code to initialize the database connection. To enable this feature, enter `SQL` followed by a space and the SQL code you want to run in the Init SQL attribute on the JDBC Data Source: Configuration: Connection Pool page in the Administration Console. If you leave this attribute blank (the default), WebLogic Server does not run any code to initialize database connections.

WebLogic Server runs this code whenever it creates a database connection for the data source, which includes at server startup, when expanding the connection pool, and when refreshing a connection.

You can use this feature to set DBMS-specific operational settings that are connection-specific or to ensure that a connection has memory or permissions to perform required actions.

Start the code with `SQL` followed by a space. For example:

```
SQL alter session set NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS'
```

or

```
SQL SET LOCK MODE TO WAIT
```

Options that you can set using `InitSQL` vary by DBMS.

Note: Init SQL is not a dynamic attribute. When you change the value for Init SQL, you must either undeploy and redeploy the data source or restart the server.

Setting Database Security Credentials

The following sections provide information on how to pass security credentials to a DBMS:

- "Types of Data Source Pools" on page 3-16
- "Using a User Name/Password" on page 3-16
- "Set Client ID On Connection" on page 3-17
- "Identity-based Connection Pooling" on page 3-17

Types of Data Source Pools

Weblogic Server provides two types of data source pools based on security privileges:

- Homogeneous—Regardless of the end user of the application, all connections in the pool use the same security credentials to access the DBMS.
- Heterogeneous—Allows applications to use a JDBC connection with a specific DBMS credential by pooling physical connections with different DBMS credentials.

This section compares methods of passing security credentials to a DBMS.

Table 3-1 Comparing Methods of Passing Security Credentials

Method	Type of Connection Pool
Using a User Name/Password	Homogeneous pool of connections.
Set Client ID On Connection	Homogeneous pool of connections.
Identity-based Connection Pooling	Heterogeneous pool of connections.

Using a User Name/Password

The simplest type of credential is to provide the connection pool a user account name and password for the DBMS. All the connections in the pool then use the same credentials to access a DBMS. See Create JDBC data sources.

Note: You can enter the password as a name-value pair in the `Properties` field (not permitted for production environments) or you can enter it in the Password field. The value in the Password field overrides any `password` value defined in the `Properties` passed to the JDBC Driver when creating physical database connections. BEA recommends that you use the Password attribute in place of the password property in the properties string because the Password value is encrypted in the configuration file (stored as the `password-encrypted` attribute in the `jdbc-driver-params` tag in the module file) and is hidden in the administration console.

Set Client ID On Connection

If the `Set Client ID On Connection` attribute is enabled on the data source, when an application requests a database connection from the data source, the WebLogic Server instance determines the current WebLogic user ID and then sets the mapped database ID as a light-weight client ID. All the connections in the pool have the same credentials to access a DBMS. Basic configuration steps are:

1. Select `Set Client ID On Connection`, see Enable Set Client ID On Connection for a JDBC data source in *Administration Console Online Help*.
2. Map the WebLogic user ID and the database ID. See Configure credential mapping for a JDBC data source in the *Administration Console Online Help*.

This feature relies on features in the JDBC driver and DBMS. It is only supported for use with Oracle and DB2 databases using a vendor extension method:

- `oracle.jdbc.OracleConnection.setClientIdentifier(String id)`
- `com.ibm.db2.jcc.DB2Connection.setDB2ClientUser(String user)`

Note: `Set Client ID On Connection` and `Enable Identity Based Connection Pooling` are mutually exclusive. If you think you need both mechanisms to pass security credentials in your application environment, create separate data sources—one for with `Set Client ID On Connection` and one with `Enable Identity Based Connection Pooling`.

Identity-based Connection Pooling

Identity-based connection pooling allows applications to use a JDBC connection with a specific DBMS credential by pooling physical connections with different DBMS credentials.

If the `Enable Identity Based Connection Pooling` attribute is enabled on the data source, when an application requests a database connection, the WebLogic Server instance selects an existing physical connection or creates a new physical connection with requested DBMS identity based on a map of WebLogic user credentials and DBMS credentials. Basic configuration steps are:

1. Select `Enable Identity Based Connection Pooling`, see "Enable identity-based connection pooling for a JDBC data source" in *Administration Console Online Help*.
2. Map WebLogic user credentials and DBMS credentials. See Configure credential mapping for a JDBC data source in the *Administration Console Online Help*.

Note: `Set Client ID On Connection` and `Enable Identity Based Connection Pooling` are mutually exclusive. If you think you need both mechanisms to pass security credentials in your application environment, create separate data sources—one for with `Set Client ID On Connection` and one with `Enable Identity Based Connection Pooling`.

How Heterogeneous Connections are Created

The following section provides information on how heterogeneous connections are created:

1. At connection pool initialization, the physical JDBC connections are created with the default DBMS credential of the data source.
2. An application tries to get a connection from a data source.
3. The current server instance credential is mapped to a DBMS credential. See "Configure credential mapping for a JDBC data source" in the *Administration Console Online Help*.

- If no match is found, the default DBMS credential is used.

  Note: The default DBMS credential should have minimum DBMS privileges, such as the ability to execute XA transactions and perform connection test operations.

- If a match is found, it is used to find physical connections matching the DBMS credential.
  - If a match is found, the connection is reserved and returned to the application.
  - If no match is found, a connection is created or reused based on the maximum capacity of the pool:
    - If the maximum capacity has not been reached, a new connection is created with the DBMS credential, reserved, and returned to the application.
    - If the pool has reached maximum capacity, based on the least recently used (LRU) algorithm, a physical connection is selected from the pool and destroyed. A new connection is created with the DBMS credential, reserved, and returned to the application.

Regardless of how physical connections are created, each physical connection in the pool has its own DBMS credential information maintained by the pool. Once a physical connection is reserved by the pool, it does not change its DBMS credential even if the current thread changes its WebLogic user credential and continues to use the same connection.

Using Identity-based Pooling with Global Transactions

When executing inside a global transaction, an application may change the credential on the current thread and get multiple JDBC connections under different credentials. However, the Identity-based Pooling feature maps multiple logical JDBC connections of a WebLogic JDBC data source inside of a global transaction into a single physical JDBC connection. This means that only one DBMS credential per WebLogic JDBC data source per WebLogic server instance is honored for a global transaction.

Using Identity-based Pooling with LLR

You must make the following changes to use Logging Last Resource (LLR) transaction optimization with Identity-based Pooling:

- You must configure a custom schema for LLR using a fully qualified LLR table name. All LLR connections will then use the named schema rather than the default schema when accessing the LLR transaction table.
- Use database specific administration tools to grant permission to access the named LLR table to all users. By default, the LLR table is created during boot by the user configured for the connection in the data source. In most cases, the database will only allow access to this user and not allow access to mapped users.

Tuning Data Source Connection Pool Options

By properly configuring the connection pool attributes in JDBC data sources in your WebLogic Server domain, you can improve application and system performance. The following sections include information about tuning options for the connection pool in a JDBC data source:

- "Increasing Performance with the Statement Cache" on page 3-20
- "Connection Testing Options for a Data Source" on page 3-23
- "Enabling Connection Creation Retries" on page 3-28
- "Enabling Connection Requests to Wait for a Connection" on page 3-29
- "Automatically Recovering Leaked Connections" on page 3-30
- "Avoiding Server Lockup with the Correct Number of Connections" on page 3-30
- "Limiting Statement Processing Time with Statement Timeout" on page 3-31
- "Using Pinned-To-Thread Property to Increase Performance" on page 3-31

Increasing Performance with the Statement Cache

When you use a prepared statement or callable statement in an application or EJB, there is considerable processing overhead for the communication between the application server and the database server and on the database server itself. To minimize the processing costs, WebLogic Server can cache prepared and callable statements used in your applications. When an application or EJB calls any of the statements stored in the cache, WebLogic Server reuses the statement stored in the cache. Reusing prepared and callable statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks.

Each connection in a data source has its own individual cache of prepared and callable statements used on the connection. However, you configure statement cache options per data source. That is, the statement cache for each connection in a data source uses the statement cache options specified for the data source, but each connection caches it's own statements. Statement cache configuration options include:

- **Statement Cache Type**—The algorithm that determines which statements to store in the statement cache. See "Statement Cache Algorithms" on page 3-20.
- **Statement Cache Size**—The number of statements to store in the cache for each connection. The default value is `10`. See "Statement Cache Size" on page 3-21.

You can use the Administration Console to set statement cache options for a data source. See "Configure the statement cache for a JDBC data source" in the *Administration Console Online Help*.

Statement Cache Algorithms

The Statement Cache Type (or algorithm) determines which prepared and callable statements to store in the cache for each connection in a data source. You can choose from the following options:

- LRU (Least Recently Used)
- Fixed

LRU (Least Recently Used)

When you select LRU (Least Recently Used, the default) as the Statement Cache Type, WebLogic Server caches prepared and callable statements used on the connection until the statement cache size is reached. When an application calls `Connection.prepareStatement()`, WebLogic Server checks to see if the statement is stored in the statement cache. If so, WebLogic

Server returns the cached statement (if it is not already being used). If the statement is not in the cache, and the cache is full (number of statements in the cache = statement cache size), Weblogic Server determines which existing statement in the cache was the least recently used and replaces that statement in the cache with the new statement.

The LRU statement cache algorithm in WebLogic Server uses an approximate LRU scheme.

Fixed

When you select FIXED as the Statement Cache Type, WebLogic Server caches prepared and callable statements used on the connection until the statement cache size is reached. When additional statements are used, they are not cached.

With this statement cache algorithm, you can inadvertently cache statements that are rarely used. In many cases, the LRU algorithm is preferred because rarely used statements will eventually be replaced in the cache with frequently used statements.

Statement Cache Size

The Statement Cache Size attribute determines the total number of prepared and callable statements to cache for each connection in each instance of the data source. By caching statements, you can increase your system performance. However, you must consider how your DBMS handles open prepared and callable statements. In many cases, the DBMS will maintain a cursor for each open statement. This applies to prepared and callable statements in the statement cache. If you cache too many statements, you may exceed the limit of open cursors on your database server.

For example, if you have a data source with 10 connections deployed on 2 servers, if you set the Statement Cache Size to 10 (the default), you may open 200 (10 x 2 x 10) cursors on your database server for the cached statements.

Usage Restrictions for the Statement Cache

Using the statement cache can dramatically increase performance, but you must consider its limitations before you decide to use it. Please note the following restrictions when using the statement cache.

There may be other issues related to caching statements that are not listed here. If you see errors in your system related to prepared or callable statements, you should set the statement cache size to `0`, which turns off statement caching, to test if the problem is caused by caching prepared statements.

Calling a Stored Statement After a Database Change May Cause Errors

Prepared statements stored in the cache refer to specific database objects at the time the prepared statement is cached. If you perform any DDL (data definition language) operations on database objects referenced in prepared statements stored in the cache, the statements may fail the next time you run them. For example, if you cache a statement such as `select * from emp` and then drop and recreate the `emp` table, the next time you run the cached statement, the statement may fail because the exact `emp` table that existed when the statement was prepared, no longer exists.

Likewise, prepared statements are bound to the data type for each column in a table in the database at the time the prepared statement is cached. If you add, delete, or rearrange columns in a table, prepared statements stored in the cache are likely to fail when run again.

These limitations depend on the behavior of your DBMS.

Using setNull In a Prepared Statement

If you cache a prepared statement that uses a `setNull` bind variable, you must set the variable to the proper data type. If you use a generic data type, as in the following example, data may be truncated or the statement may fail when it runs with a value other than null.

```
java.sql.Types.Long sal=null
.
.
.
if (sal == null)
   setNull(2,int)//This is incorrect
else
   setLong(2,sal)
```

Instead, use the following:

```
if (sal == null)
   setNull(2,long)//This is correct
else
   setLong(2,sal)
```

Statements in the Cache May Reserve Database Cursors

When WebLogic Server caches a prepared or callable statement, the statement may open a cursor in the database. If you cache too many statements, you may exceed the limit of open cursors for a connection. To avoid exceeding the limit of open cursors for a connection, you can change the

limit in your database management system or you can reduce the statement cache size for the data source.

Connection Testing Options for a Data Source

To make sure that the database connections in a data source remain healthy, you should periodically test the connections. WebLogic Server includes two basic types of testing:

- Automatic testing that you configure with options on the data source so that WebLogic Server makes sure that database connections remain healthy.
- Manual testing that you can do to trouble-shoot a data source.

The following section discusses automatic connection testing options. For more information about manual connection testing, see "Testing Data Sources and Database Connections" on page 7-1.

To configure automatic testing options for a data source, you set the following options either through the Administration Console or through WLST using the JDBCConnectionPoolParamsBean:

- Test Frequency—(TestFrequencySeconds in the JDBCConnectionPoolParamsBean) Use this attribute to specify the number of seconds between tests of unused connections. WebLogic Server tests unused connection and reopens any faulty connections. You must also set the Test Table Name.
- Test Reserved Connections—(TestConnectionsOnReserve in the JDBCConnectionPoolParamsBean) Select this option to test each connection before giving to a client. This may add a slight delay to the request, but it guarantees that the connection is healthy. You must also set a Test Table Name.
- Test Table Name—(TestTableName in the JDBCConnectionPoolParamsBean) Use this attribute to specify a table name to use in a connection test. You can also specify SQL code to run in place of the standard test by entering `SQL` followed by a space and the SQL code you want to run as a test. Test Table Name is required to enable any database connection testing.
- Seconds to Trust an Idle Pool Connection—(SecondsToTrustAnIdlePoolConnection in the JDBCConnectionPoolParamsBean) Use this option to The number of seconds within a connection use that WebLogic Server trusts that the connection is still viable and will skip the connection test, either before delivering it to an application or during the periodic connection testing process. This option is an optimization that minimizes the performance

impact of connection testing, especially during heavy traffic. See "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection" on page 3-26.

See the JDBC Data Source: Configuration: Connection Pool page in the Administration Console or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about these options.

For instructions to set connection testing options, see "Configure testing options for a JDBC data source" in the *Administration Console Online Help*.

Database Connection Testing Semantics

When WebLogic Server tests database connections in a datasource, it reserves a connection from the datasource, runs a small query on the connection, then returns the connection to the pool in the data source. The server instance tracks statistics on the pool status, including the amount of time a required to complete a connection test, the number of connections waiting for a connection, and the number of connections being tested. The history of recent test connection behavior is used to calculate the amount of time the server instance waits until a connection test is determined to have failed. If a thread appears to be taking longer than normal to connect, the server instance delays testing on other threads until:

- The test completes sucessfully and pending tests on other threads resumes or
- The connection test fails.

The query used in testing is determined by the value in Test Table Name. If the value is a table name, the query is `select 1 from table_name`. If Test Table Name includes a full query starting with `SQL` followed by space and the query, WebLogic Server uses that query when testing database connections.

If a connection fails the test, WebLogic Server closes and recreates the connection, and then tests the new connection.

Details about the semantics of connection testing is discussed in the following sections:

- "Connection Testing When Database Connections are Created" on page 3-25
- "Periodic Connection Testing" on page 3-25
- "Testing Reserved Connections" on page 3-25
- "Minimized Connection Test Delay After Database Connectivity Loss" on page 3-25
- "Minimized Connection Request Delay After Connection Test Failures" on page 3-26

- "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection" on page 3-26

Connection Testing When Database Connections are Created

When connections are created in a data source, WebLogic Server tests each connection using the query defined by the value in Test Table Name. Connections are created when a data source is deployed, either at server startup or when creating a data source, when increasing capacity to meet demand for connections, or when recreating a connection that failed a connection test.

The purpose of this testing is to ensure that new connections are viable and ready for use when an application requests a connection.

Periodic Connection Testing

If Test Frequency is greater than 0, WebLogic Server periodically tests database connections that are not in use in the data source. The test is based on the query defined in Test Table Name. If a connection fails the test, WebLogic Server closes the connection, recreates the connection, and tests the new connection before returning it to the pool.

Testing Reserved Connections

When Test Connections On Reserve is enabled, when your application requests a connection from the data source, WebLogic Server tests the connection using the query specified in Test Table Name before giving the connection to the application.

Testing reserved connections can cause a delay in satisfying connection requests, but it makes sure that the connection is viable when the application gets the connection. You can minimize the impact of testing reserved connections by tuning Seconds to Trust an Idle Pool Connection. See "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection" on page 3-26

Minimized Connection Test Delay After Database Connectivity Loss

When connectivity to the DBMS is lost, even if only momentarily, some or all of the JDBC connections in a data source typically become defunct. If the data source is configured to test connections on reserve, when an application requests a database connection, WebLogic Server tests the connection, discovers that the connection is dead, and tries to replace it with a new connection to satisfy the request. Ordinarily, when the DBMS comes back online, the refresh process succeeds. However, in some cases and for some modes of failure, testing a dead connection can impose a long delay.

To minimize this delay, WebLogic data sources include logic that considers *all* connections in the data source as dead after a number of consecutive test failures, and closes all connections in the data source. After all connections are closed, when an application requests a connection, the data source creates a connection without first having to test a dead connection. This behavior minimizes the delay for connection requests following the data source's connection pool flush.

WebLogic Server determines the number of test failures before closing all connections based on the Test Frequency setting for the data source:

- If Test Frequency is greater than 0, the number of test failures before closing all connections is set to 2.
- If Test Frequency is set to 0 (periodic testing is disabled), the number of test failures before closing all connections is set to 25% of the Maximum Capacity for the data source.

Minimized Connection Request Delay After Connection Test Failures

If your DBMS becomes and remains unavailable, the data source will persistently test and try to replace dead connections while trying to satisfy connection requests. This behavior is beneficial because it enables the data source to react immediately when the database becomes available. However, testing a dead database connection can take as long as the network timeout, and can cause a long delay for clients.

To minimize this delay, the WebLogic data sources include logic that disables the data source after 2 consecutive failures to replace a dead connection. When an application requests a connection from a disabled data source, WebLogic Server throws a `PoolDisabledSQLException immediately` to notify the client that a connection is not available.

For data sources that are disabled in this manner, WebLogic Server periodically runs a refresh process. The refresh process does the following:

- The server instance executes a health check on the database server every 5 seconds. This setting is not configurable.
- If the server instance recognizes that the database was recovered, it creates a new database connection and enables the data source.

You can also manually enable the data source using the Administration Console.

Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection

Database connection testing during heavy traffic can reduce application performance. To minimize the impact of connection testing, you can set the Seconds To Trust An Idle Pool

Connection attribute in the JDBC data source configuration to trust recently-used or recently-tested database connections and skip the connection test.

If Test Reserved Connections is enabled on your data source, when an application requests a database connection, WebLogic Server tests the database connection before giving it to the application. If the request is made within the time specified for Seconds to Trust an Idle Pool Connection since the connection was tested or successfully used and returned to the data source, WebLogic Server skips the connection test before delivering it to an application.

If Test Frequency is greater than 0 for your data source (periodic testing is enabled), WebLogic Server also skips the connection test if the connection was successfully used and returned to the data source within the time specified for Seconds to Trust an Idle Pool Connection.

For instructions to set Seconds to Trust an Idle Pool Connection, see "Configure testing options for a JDBC data source" in the *Administration Console Online Help*.

Seconds to Trust an Idle Pool Connection is a tuning feature that can improve application performance by minimizing the delay caused by database connection testing, especially during heavy traffic. However, it can reduce the effectiveness of connection testing, especially if the value is set too high. The appropriate value depends on your environment and the likelihood that a connection will become defunct.

Database Connection Testing Configuration Recommendations

You should set connection testing attributes so that they best fit your environment. For example, if your application cannot tolerate database connection failures, you should set Seconds to Trust an Idle Pool Connection to 0 and make sure Test Reserved Connections is enabled so that WebLogic Server will test every connection before giving it to an application. If your application is more sensitive delays in getting a connection from the data source and can tolerate an occasional connection failure, you should set Seconds to Trust an Idle Pool Connection to a higher number, set Test Frequency to a low number, and enable Test Reserved Connections. With these settings, your application would rely more on testing connections in the pool when they are not in use rather than when an application requests a connection.

Default Test Table Name

When you create a data source using the Administration Console, the Administration Console automatically sets the Test Table Name attribute for a data source based on the DBMS that you select. The Test Table Name attribute is used in connection testing which is optionally performed periodically or when you create or reserve a connection, depending on how you configure the testing options. For database tests to succeed, the database user used to create database

connections in the data source must have access to the database table. If not, you should either grant access to the user (make this change in the DBMS) or change the Test Table Name attribute to the name of a table to which the user does have access (make this change in the WebLogic Server Administration Console).

Table 3-2 Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
Adabas for z/OS	SQL call shadow_adabas('select * from employees')
Cloudscape	SQL SELECT 1
DB2	SQL SELECT COUNT(*) FROM SYSIBM.SYSTABLES
FirstSQL	SQL SELECT 1
IMS/TM for z/OS	SQL call shadow_ims('otm','/dis','cctl')
Informix	SQL SELECT COUNT(*) FROM SYSTABLES
Microsoft SQL Server	SQL SELECT 1
MySQL	SQL SELECT 1
Oracle	SQL SELECT 1 FROM DUAL
PointBase	SQL SELECT COUNT(*) FROM SYSTABLES
PostgreSQL	SQL SELECT 1
Progress	SQL SELECT COUNT(*) FROM SYSTABLES
Sybase	SQL SELECT 1

Enabling Connection Creation Retries

WebLogic JDBC data sources include the Connection Creation Retry Frequency option (ConnectionCreationRetryFrequencySeconds in the JDBCConnectionPoolParamsBean) that you can use to specify the number of seconds between attempts to establish connections to the database. If set and if the database is unavailable when the data source is created, WebLogic Server attempts to create connections in the pool again after the number of seconds you specify, and will continue to attempt to create the connections until it succeeds. This option applies to connections created when the data source is created at server startup or when the data source is

deployed or if the initial capacity is increased. It does *not* apply to connections created for pool expansion or to replace a defunct connection in the pool.

By default, Connection Creation Retry Frequency is 0 seconds. When the value is set to 0, connection creation retries is disabled and data source creation fails if the database is unavailable.

See the JDBC Data Source: Configuration: Connection Pool page in the Administration Console or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about this option.

Enabling Connection Requests to Wait for a Connection

JDBC data sources have two attributes that you can set to enable connection requests to wait for a connection from a data source: Connection Reserve Timeout (`ConnectionReserveTimeoutSeconds`) and Maximum Waiting for Connection (`HighestNumWaiters`). You use these two attributes together to enable connection requests to wait for a connection without disabling your system by blocking too many threads.

See the JDBC Data Source: Configuration: Connection Pool page in the Administration Console or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about these options.

Also see "Enable connection requests to wait for a connection" in the *Administration Console Online Help*.

Connection Reserve Timeout

When an application requests a connection from a data source, if all connections in the data source are in use and if the data source has expanded to its maximum capacity, the application will get a Connection Unavailable SQL Exception. To avoid this, you can configure the Connection Reserve Timeout value (in seconds) so that connection requests will wait for a connection to become available. After the Connection Reserve Timeout has expired, if no connection becomes available, the request will fail and the application will get a `PoolLimitSQLException` exception.

If you set Connection Reserve Timeout to `-1`, a connection request will timeout immediately if there is no connection available. If you set Connection Reserve Timeout to `0`, a connection request will wait indefinitely. The default value is 10 seconds.

See "Enable connection requests to wait for a connection" in the *Administration Console Online Help*.

Limiting the Number of Waiting Connection Requests

Connection requests that wait for a connection block a thread. If too many connection requests concurrently wait for a connection and block threads, your system performance can degrade. To avoid this, you can set the Maximum Waiting for Connection (`HighestNumWaiters`) attribute, which limits the number connection requests that can concurrently wait for a connection.

If you set Maximum Waiting for Connection (`HighestNumWaiters`) to `MAX-INT` (the default), there is effectively no bound on how many connection requests can wait for a connection. If you set Maximum Waiting for Connection to `0`, connection requests cannot wait for a connection. If the maximum number of requests has been met, a `SQLException` is thrown when an application requests a connection.

Automatically Recovering Leaked Connections

A leaked connection is a connection that was not properly returned to the connection pool in the data source. To automatically recover leaked connections, you can specify a value for Inactive Connection Timeout on the JDBC Data Source: Configuration: Connection Pool page in the Administration Console. When you set a value for Inactive Connection Timeout, WebLogic Server will forcibly return a connection to the data source when there is no activity on a reserved connection for the number of seconds that you specify. When set to `0` (the default value), this feature is turned off.

See the JDBC Data Source: Configuration: Connection Pool page in the Administration Console or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about this option.

Note that the actual timeout could exceed the configured value for Inactive Connection Timeout. The internal data source maintenance thread runs every 5 seconds. When it reaches the Inactive Connection Timeout (for example 30 seconds), it checks for inactive connections. To avoid timing out a connection that was reserved just before the current check or just after the previous check, the server gives an inactive connection a "second chance." On the next check, if the connection is still inactive, the server times it out and forcibly returns it to the data source. On average, there could be a delay of 50% more than the configured value.

Avoiding Server Lockup with the Correct Number of Connections

When your applications attempt to get a connection from a data source in which there are no available connections, the data source throws an exception stating that a connection is not

available in the data source. To avoid this error, make sure your data source can expand to the size required to accommodate your peak load of connection requests. To increase the maximum number of connections available in the data source, increase the value for Maximum Capacity for the data source on the JDBC Data Source: Configuration: Connection Pool page in the Administration Console.

Limiting Statement Processing Time with Statement Timeout

With the Statement Timeout option on a JDBC data source, you can limit the amount of time that a statement takes to execute on a database connection reserved from the data source. When you set a value for Statement Timeout, WebLogic Server passes the time specified to the JDBC driver using the `java.sql.Statement.setQueryTimeout()` method. If your JDBC driver does not support this method, it may throw an exception and the timeout value is ignored.

When Statement Timeout is set to -1, (the default) statements do not timeout.

See the JDBC Data Source: Configuration: Connection Pool page in the Administration Console or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about this option.

Using Pinned-To-Thread Property to Increase Performance

To minimize the time it takes for an application to reserve a database connection from a data source and to eliminate contention between threads for a database connection, you can add the `Pinned-To-Thread` property in the connection Properties list for the data source, and set its value to `true`.

When `Pinned-To-Thread` is enabled, WebLogic Server pins a database connection from the data source to an execution thread the first time an application uses the thread to reserve a connection. When the application finishes using the connection and calls `connection.close()`, which otherwise returns the connection to the data source, WebLogic Server keeps the connection with the execute thread and does not return it to the data source. When an application subsequently requests a connection using the same execute thread, WebLogic Server provides the connection already reserved by the thread. There is no locking contention on the data source that occurs when multiple threads attempt to reserve a connection at the same time and there is no contention for threads that attempt to reserve the same connection from a limited number of database connections.

Note: In this release, the Pinned-To-Thread feature does not work with multi data sources, Oracle RAC, and IdentityPool. These features rely on the ability to return a connection to the connection pool and reacquire it if there is a connection failure or connection identity does not match.

See JDBC Data Source: Configuration: Connection Pool in the *Administration Console Online Help*.

Changes to Connection Pool Administration Operations When PinnedToThread is Enabled

Because the nature of connection pooling behavior is changed when `PinnedToThread` is enabled, some connection pool attributes or features behave differently or are disabled to suit the behavior change:

- **Maximum Capacity** is ignored. The number of connections in a connection pool equals the greater of either the initial capacity or the number of connections reserved from the connection pool.
- **Shrinking** does not apply to connection pools with `PinnedToThread` enabled because connections are never returned to the connection pool. Effectively, they are always reserved.
- When you **Reset** a connection pool, the reset connections from the connection pool are marked as Test Needed. The next time each connection is reserved, WebLogic Server tests the connection and recreates it if necessary. Connections are not tested synchronously when you reset the connection pool. This feature requires that Test Connections on Reserve is enabled and a Test Table Name or query is specified.

Additional Database Resource Costs When PinnedToThread is Enabled

When `PinnedToThread` is enabled, the maximum capacity of the connection pool (maximum number of database connections created in the connection pool) becomes the number of execute threads used to request a connection multiplied by the number of concurrent connections each thread reserves. This may exceed the Maximum Capacity specified for the connection pool. You may need to consider this larger number of connections in your system design and ensure that your database allows for additional associated resources, such as open cursors.

Also note that connections are never returned to the connection pool, which means that the connection pool can never shrink to reduce the number of connections and associated resources in use. You can minimize this cost by setting an additional driver parameter `onePinnedConnectionOnly`. When `onePinnedConnectionOnly`=*true*, only the first

connection requested is pinned to the thread. Any additional connections required by the thread are taken from and returned to the connection pool as needed. Set `onePinnedConnectionOnly` using the `Properties` attribute, for example:

```
   Properties="PinnedToThread=true;onePinnedConnectionOnly=true;user=examp
les"
```

If your system can handle the additional resource requirements, BEA recommends that you use the `PinnedToThread` option to increase performance.

If your system cannot handle the additional resource requirements or if you see database resource errors after enabling `PinnedToThread`, BEA recommends *not* using `PinnedToThread`.

Deploying Data Sources on Servers and Clusters

To deploy a data source to a cluster or server, you select the server or cluster as a deployment target. When a data source is deployed on a server, WebLogic Server creates an instance of the data source on the server, including the pool of database connections in the data source. When you deploy a data source to a cluster, WebLogic Server creates an instance of the data source on each server in the cluster.

For instructions, see "Target JDBC data sources" in the *Administration Console Online Help*.

Minimizing Server Startup Hang Caused By an Unresponsive Database

On server startup, WebLogic Server attempts to create database connections in the data sources deployed on the server. If a database is unreachable, server startup may hang in the STANDBY state for a long period of time. This is due to WebLogic Server threads that hang inside the JDBC driver code waiting for a reply from the database server. The duration of the hang depends on the JDBC driver and the TCP/IP timeout setting on the WebLogic Server machine.

To work around this issue, WebLogic Server includes the JDBCLoginTimeoutSeconds attribute on the ServerMBean. When you set a value for this attribute, the value is passed into `java.sql.DriverManager.setLoginTimeout()`. If the JDBC driver being used to create database connections implements the setLoginTimeout method, attempts to create database connections will wait only as long as the timeout specified.

Security for JDBC Data Sources

The following sections provide information on how WebLogic Server uses roles and policies to secure JDBC data sources:

- "Setting Security Policies for JDBC Resources" on page 3-34
- "Security Roles for JDBC MBeans" on page 3-34

Setting Security Policies for JDBC Resources

You can optionally restrict access to JDBC data sources. In WebLogic Server, security policies answer the question "who has access" to a WebLogic resource. A security policy is created when you define an association between a WebLogic resource and a user, group, or role. A WebLogic resource has no protection until you assign it a security policy. For instructions on how to set up security for all WebLogic Server resources, see "Use roles and policies to secure resources" in *Administration Console Online Help*. For more information about securing server resources, see *Securing WebLogic Resources*.

You can protect JDBC operations by assigning Administrator methods which can limit the actions that an administrator may take upon a JDBC data source. See "Java DataBase Connectivity (JDBC) Resources" in *Securing WebLogic Resources Using Roles and Policies*.

Security Roles for JDBC MBeans

JDBC MBeans allow only the Admin and Deployer roles. The following section provides information on the security roles defined for JDBC MBeans:

- "JDBC Domain Configuration MBeans" on page 3-34
- "JDBC System Module MBeans" on page 3-35

See "Default Security Policies for MBeans" in *WebLogic Server MBean Reference* for information on default security settings for WebLogic Server.

JDBC Domain Configuration MBeans

The following domain configuration JDBC MBeans that have settings that override the default security settings.

- JDBCConnectionPoolMBean (deprecated)
- JDBCDataSourceFactoryMBean (deprecated)

- JDBCDataSourceMBean (deprecated)
- JDBCMultiPoolMBean (deprecated)
- JDBCSystemResourceMBean
- JDBCTxDataSourceMBean (deprecated)

See "Domain Configuration MBeans" in *WebLogic Server MBean Reference*.

JDBC System Module MBeans

The following system module JDBC MBeans that have settings that override the default security settings.

- JDBCConnectionPoolParamsBean
- JDBCDataSourceBean
- JDBCDataSourceParamsBean
- JDBCDriverParamsBean
- JDBCPropertiesBean
- JDBCPropertyBean
- JDBCXAParamsBean

See "System Module MBeans" in *WebLogic Server MBean Reference*.

JDBC Data Source Factories (Deprecated)

In previous releases of WebLogic Server, application-scoped JDBC connection pools relied on JDBC data source factories to provide default connection pool values. JDBC data source factories are deprecated in WebLogic Server 9.2 and are included in the release for backward compatibility only. Application-scoped JDBC connection pools are replaced by JDBC application modules. For more information, see "Application Scoping for a Packaged JDBC Module" on page A-7.

CHAPTER 4

Configuring JDBC Multi Data Sources

A *multi data source* is an abstraction around a group of data sources that provides load balancing or failover processing between the data sources associated with the multi data source. Multi data sources are bound to the JNDI tree or local application context just like data sources are bound to the JNDI tree. Applications lookup a multi data source on the JNDI tree or in the local application context (`java:comp/env`) just like they do for data sources, and then request a database connection. The multi data source determines which data source to use to satisfy the request depending on the algorithm selected in the multi data source configuration: load balancing or failover.

This section includes the following information:

- "Multi Data Source Features" on page 4-2
- "Creating and Configuring Multi Data Sources" on page 4-3
- "Choosing the Multi Data Source Algorithm" on page 4-3
- "Multi Data Source Fail-Over Limitations and Requirements" on page 4-4
- "Multi Data Source Failover Enhancements" on page 4-5
- "Deploying JDBC Multi Data Sources on Servers and Clusters" on page 4-11

Multi Data Source Features

A multi data source can be thought of as a pool of data sources. Multi data sources are best used for failover or load balancing between nodes of a highly available database system, such as redundant databases or Oracle Real Application Clusters (RAC).

The data source member list for a Multi data source supports dynamic updates. This allows environments, such as those using Oracle RAC, to add and remove database nodes and corresponding data sources without redeployment and provide the ability to:

- Grow and shrink RAC clusters in response to throughput. See "Adding a Database Node" on page 4-3.
- Shutdown RAC nodes for maintenance. See "Removing a Database Node" on page 4-2.

See "Using Multi Data Sources with Oracle RAC" on page B-13.

Note: Multi data sources do not provide any synchronization between databases. It is assumed that database synchronization is handled properly outside of WebLogic Server so that data integrity is maintained.

Removing a Database Node

You can remove a database node and corresponding data sources without redeployment. This capability provides you the ability to shutdown a node for maintenance or shrink a cluster. Use the following high-level steps to shutdown a database node:

Note: Failure to follow these step may cause transaction roll-backs.

1. Remove the data source from the multi data source. See "Add or remove data sources in a JDBC multi data source" in *Administration Console Online Help.*
2. Gracefully suspend the data source. See "Suspend JDBC data sources" in *Administration Console Online Help.*
3. When all transactions have completed, shut down the data source. See "Shut down JDBC data sources" in *Administration Console Online Help.*
4. Shut down the database node.

Adding a Database Node

You can add a database node and corresponding data sources without redeployment. This capability provides you the ability to start a node after maintenance or grow a cluster. Use the following high-level steps to add a database node:

1. Restart the database node.
2. Restart the data source. See "Start JDBC data sources" in *Administration Console Online Help*.
3. Add the data source back to the multi data source. See "Add or remove data sources in a JDBC multi data source" in *Administration Console Online Help.*

Creating and Configuring Multi Data Sources

You create a multi data source by first creating data sources, then creating the multi data source using the Administration Console or the WebLogic Scripting Tool and then assigning the data sources to the multi data source.

For instructions to create a multi data source, see "Configure JDBC multi data sources" in the *Administration Console Online Help*.

For information about the configuration files created when configuring a multi data source, see "Understanding JDBC Resources in WebLogic Server" on page 2-1. Also see "Creating a JDBC Multi Data Source Module" on page A-5.

Choosing the Multi Data Source Algorithm

Before you set up a multi data source, you need to determine the primary purpose of the multi data source—failover or load balancing. You can choose the algorithm that corresponds with your requirements.

Failover

The Failover algorithm provides an ordered list of data sources to use to satisfy connection requests. Normally, every connection request to this kind of multi data source is served by the first data source in the list. If a database connection test fails and the connection cannot be replaced, or if the data source is suspended, a connection is sought sequentially from the next data source on the list.

Note: This algorithm relies on Test Reserved Connections (`TestConnectionsOnReserve`) on the data source to test a connection in the first data source to see if the data source is healthy. If the connection fails the test, the multi data source uses a connection from the next data source listed in the multi data source. See "Connection Testing Options for a Data Source" on page 3-23 for information about configuring `TestConnectionsOnReserve`.

JDBC is a highly stateful client-DBMS protocol, in which the DBMS connection and transactional state are tied directly to the socket between the DBMS process and the client (driver). For this reason, failover of a connection while it is in use is not supported.

Load Balancing

Connection requests to a load-balancing multi data source are served from any data source in the list. The multi data source selects data sources to use to satisfy connection requests using a round-robin scheme. When the multi data source provides a connection, it selects a connection from the data source listed just after the last data source that was used to provide a connection. Multi data sources that use the Load Balancing algorithm also fail over to the next data source in the list if a database connection test fails and the connection cannot be replaced, or if the data source is suspended.

Multi Data Source Fail-Over Limitations and Requirements

WebLogic Server provides the Failover algorithm for multi data sources so that if a data source fails (for example, if the database management system crashes), your system can continue to operate. However, you must consider the following limitations and requirements when configuring your system.

Test Connections on Reserve to Enable Fail-Over

Data sources rely on the Test Reserved Connections (`TestConnectionsOnReserve`) feature on the data source to know when database connectivity is lost. Testing reserved connections must be enabled (the default) for the data sources within the multi data source. WebLogic Server will test each connection before giving it to an application. With the Failover algorithm, the multi data source uses the results from connection test to determine when to fail over to the next data source in the multi data source. After a test failure, the data source attempts to recreate the connection. If that attempt fails, the multi data source fails over to the next data source.

No Fail-Over for In-Use Connections

It is possible for a connection to fail after being reserved, in which case your application must handle the failure. WebLogic Server cannot provide fail-over for connections that fail while being used by an application. Any failure while using a connection requires that you restart the transaction and provide code to handle such a failure.

Multi Data Source Failover Enhancements

The following enhancements improve failover processing for multi data sources:

- Connection request routing enhancements to avoid requesting a connection from an automatically disabled (dead) data source within a multi data source. See "Connection Request Routing Enhancements When a Data Source Fails."
- Automatic failback on recovery of a failed data source within a multi data source. See "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source."
- Failover for busy data sources within a multi data sources. See "Enabling Failover for Busy Data Sources in a Multi Data Source."
- Failover callbacks for multi data sources with the Failover algorithm. See "Controlling Multi Data Source Failover with a Callback."
- Failback callbacks for multi data sources with either algorithm. See "Controlling Multi Data Source Failback with a Callback."

Connection Request Routing Enhancements When a Data Source Fails

To improve performance when a data source within a multi data source fails, WebLogic Server automatically disables the data source when a pooled connection fails a connection test. After a data source is disabled, WebLogic Server does not route connection requests from applications to the data source. Instead, it routes connection requests to the next available data source listed in the multi data source.

This feature requires that data source testing options are configured for all data sources in a multi data source, specifically Test Table Name and Test Reserved Connections. See "Connection Testing Options for a Data Source" on page 3-23.

If a callback handler is registered for the multi data source, WebLogic Server calls the callback handler before failing over to the next data source in the list. See "Controlling Multi Data Source Failover with a Callback" on page 4-7 for more details.

Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source

After a data source is automatically disabled because a connection failed a connection test, the multi data source periodically tests a connection from the disabled data source to determine when the data source (or underlying database) is available again. When the data source becomes available, the multi data source automatically re-enables the data source and resumes routing connection requests to the data source, depending on the multi data source algorithm and the position of the data source in the list of included data sources. Frequency of these tests is controlled by the Test Frequency Seconds attribute of the multi data source. The default value for Test Frequency is 120 seconds, so if you do not specifically set a value for the option, the multi data source will test disabled data sources every 120 seconds. See JDBC Multi Data Source: Configuration: General in the *Administration Console Online Help*.

WebLogic Server does not test and automatically re-enable data sources that you manually disable. It only tests data sources that are automatically disabled.

If a callback handler is registered for the multi data source, WebLogic Server calls the callback handler before re-enabling the data source. See "Controlling Multi Data Source Failback with a Callback" on page 4-9 for more details.

Enabling Failover for Busy Data Sources in a Multi Data Source

By default, for multi data sources with the Failover algorithm, when the number of requests for a database connection exceeds the number of available connections in the current data source in the multi data source, subsequent connection requests fail.

To enable the multi data source to failover when all connections in the current data source are in use, you can enable the Failover Request if Busy option on the JDBC Multi Data Source: Configuration: General page in the Administration Console. (Also available as the `FailoverRequestIfBusy` attribute in the JDBCDataSourceParamsBean.). If enabled (set to `true`), when all connections in the current data source are in use, application requests for connections will be routed to the next available data source within the multi data source. When disabled (set to `false`, the default), connection requests do not failover.

If a `ConnectionPoolFailoverCallbackHandler` is included in the multi data source configuration, WebLogic Server calls the callback handler before failing over. See "Controlling Multi Data Source Failover with a Callback" on page 4-7 for more details.

Controlling Multi Data Source Failover with a Callback

You can register a callback handler with WebLogic Server that controls when a multi data source with the Failover algorithm fails over connection requests from one JDBC data source in the multi data source to the next data source in the list.

You can use callback handlers to control if or when the failover occurs so that you can make any other system preparations before the failover, such as priming a database or communicating with a high-availability framework.

Callback handlers are registered via the Failover Callback Handler attribute of the multi data source and are registered per multi data source. You must register the callback handler for each multi data source to which you want the callback handler to apply. And you can register different callback handlers for each multi data source in your domain.

Callback Handler Requirements

A callback handler used to control the failover and failback within a multi data source must include an implementation of the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface. When the multi data source needs to failover to the next data source in the list or when a previously disabled data source becomes available, WebLogic Server calls the `allowPoolFailover()` method in the `ConnectionPoolFailoverCallback` interface, and passes a value for the three parameters, `currPool`, `nextPool`, and `opcode`, as defined below. WebLogic Server then waits for the return from the callback handler before completing the task.

Your application must return OK, RETRY_CURRENT, or DONOT_FAILOVER as defined below. The application should handle failover and failback cases.

See the Javadoc for the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface for more details.

Note: Failover callback handlers are optional.If no callback handler is specified in the multi data source configuration, WebLogic Server proceeds with the operation (failing over or re-enabling the disabled data source).

Callback Handler Configuration

There are two multi data source configuration attributes associated with the failover and failback functionality:

- Failover Callback Handler (`ConnectionPoolFailoverCallbackHandler`)—To register a failover callback handler for a multi data source, you add a value for this attribute to the multi data source configuration. The value must be an absolute name, such as `com.bea.samples.wls.jdbc.MultiDataSourceFailoverCallbackApplication`. You can set the Failover Callback Handler using the Administration Console (see "Register a failover callback handler") or on the JDBCDataSourceParamsBean for the multi data source using WLST.
- Test Frequency (`TestFrequencySeconds`)—To control how often the multi data source checks disabled (dead) data sources to see if they are now available. See "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source" on page 4-6 for more details.

How It Works—Failover

WebLogic Server attempts to failover connection requests to the next data source in the list when the current data source fails a connection test or, if you enabled `FailoverRequestIfBusy`, when all connections in the current data source are busy.

To enable the callback feature, you register the callback handler with Weblogic Server using Failover Callback Handler in the multi data source configuration.

With the Failover algorithm, connection requests are served from the first data source in the list. If a connection from that data source fails a connection test, WebLogic Server marks the data source as dead and disables it. If a callback handler is registered, WebLogic Server calls the callback handler, passing the following information, and waits for a return:

- `currPool`—For failover, this is the name of data source currently being used to supply database connections. This is the "failover from" data source.
- `nextPool`—The name of next available data source listed in the multi data source. For failover, this is the "failover to" data source.
- `opcode`—A code that indicates the reason for the call:
  - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current data source is dead and has disabled it.

  – `OPCODE_CURR_POOL_BUSY`—All database connections in the data source are in use. (Requires `FailoverIfBusy=true` in the multi data source configuration. See "Enabling Failover for Busy Data Sources in a Multi Data Source" on page 4-6.)

Failover is synchronous with the connection request: Failover occurs only when WebLogic Server is attempting to satisfy a connection request.

The return from the callback handler can indicate one of three options:

- `OK`—proceed with the operation. In this case, that means to failover to the next data source in the list.
- `RETRY_CURRENT`—Retry the connection request with the current data source.
- `DONOT_FAILOVER`—Do not retry the current connection request and do not failover. WebLogic Server will throw a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

WebLogic Server acts according to the value returned by the callback handler.

If the secondary data sources fails, WebLogic Server calls the callback handler again, as in the previous failover, in an attempt to failover to the next available data source in the multi data source, if there is one.

Note: WebLogic Server does *not* call the callback handler when you manually disable a data source.

For multi data sources with the Load-Balancing algorithm, WebLogic Server does not call the callback handler when a data source is disabled. However, it does call the callback handler when attempting to re-enable a disabled data source. See the following section for more details.

Controlling Multi Data Source Failback with a Callback

If you register a failover callback handler for a multi data source, WebLogic Server calls the same callback handler when re-enabling a data source that was automatically disabled. You can use the callback to control if or when the disabled data source is re-enabled so that you can make any other system preparations before the data source is re-enabled, such as priming a database or communicating with a high-availability framework.

See the following sections for more details about the callback handler:

- "Callback Handler Requirements" on page 4-7
- "Callback Handler Configuration" on page 4-8

How It Works—Failback

WebLogic Server periodically checks the status of data sources in a multi data source that were automatically disabled. (See "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source" on page 4-6.) If a disabled data source becomes available and if a failover callback handler is registered, WebLogic Server calls the callback handler with the following information and waits for a return:

- `currPool`—For failback, this is the name of the data source that was previously disabled and is now available to be re-enabled.
- `nextPool`—For failback, this is null.
- `opcode`—A code that indicates the reason for the call. For failback, the code is always `OPCODE_REENABLE_CURR_POOL`, which indicates that the data source named in `currPool` is now available.

Failback, or automatically re-enabling a disabled data source, differs from failover in that failover is *synchronous* with the connection request, but failback is *asynchronous* with the connection request.

The callback handler can return one of the following values:

- `OK`—proceed with the operation. In this case, that means to re-enable the indicated data source. WebLogic Server resumes routing connection requests to the data source, depending on the multi data source algorithm and the position of the data source in the list of included data sources.
- `DONOT_FAILOVER`—Do not re-enable the `currPool` data source. Continue to serve connection requests from the data source(s) in use.

WebLogic Server acts according to the value returned by the callback handler.

If the callback handler returns DONOT_FAILOVER, WebLogic Server will attempt to re-enable the data source during the next testing cycle as determined by the `TestFrequencySeconds` attribute in the multi data source configuration, and will call the callback handler as part of that process.

The order in which data sources are listed in a multi data source is very important. A multi data source with the Failover algorithm will always attempt to serve connection requests from the first available data source in the list of data sources in the multi data source. Consider the following scenario:

1. `MultiDataSource_1` uses the Failover algorithm, has a registered `ConnectionPoolFailoverCallbackHandler`, and includes three data sources: `DS1`, `DS2`, and `DS3`, listed in that order.
2. `DS1` becomes disabled, so `MultiDataSource_1` fails over connection requests to `DS2`.
3. `DS2` then becomes disabled, so `MultiDataSource_1` fails over connection requests to `DS3`.
4. After some time, `DS1` becomes available again and the callback handler allows WebLogic Server to re-enable the data source. Future connection requests will be served by `DS1` because `DS1` is the first data source listed in the multi data source.
5. If `DS2` subsequently becomes available and the callback handler allows WebLogic Server to re-enable the data source, connection requests will continue to be served by `DS1` because `DS1` is listed before `DS2` in the list of data sources.

Deploying JDBC Multi Data Sources on Servers and Clusters

All data sources used by a multi data source to satisfy connection requests must be deployed on the same servers and clusters as the multi data source. A multi data source always uses a data source deployed on the same server to satisfy connection requests. Multi data sources do not route connection requests to other servers in a cluster or in a domain.

To deploy a multi data source to a cluster or server, you select the server or cluster as a deployment target. When a multi data source is deployed on a server, WebLogic Server creates an instance of the multi data source on the server. When you deploy a multi data source to a cluster, WebLogic Server creates an instance of the multi data source on each server in the cluster.

For instructions, see "Target and deploy JDBC multi data sources" in the *Administration Console Online Help*.

CHAPTER 5

Using Third-Party JDBC Drivers with WebLogic Server

WebLogic Server works with third-party JDBC drivers that offer the following functionality:

- Are thread-safe
- Can implement transactions using standard JDBC statements

Third-party JDBC drivers that do not implement Serializable or Remote interfaces cannot pass objects to a remote client application.

This section describes how to set up and use third-party JDBC drivers with WebLogic Server. It includes the following sections:

- "Third-Party JDBC Drivers Installed with WebLogic Server" on page 5-1
- "Setting the Environment for a Type-4 Third-Party JDBC Driver" on page 5-2
- "Globalization Support for the Oracle 10g Thin Driver" on page 5-3
- "Using the Oracle Thin Driver in Debug Mode" on page 5-3

Third-Party JDBC Drivers Installed with WebLogic Server

The following third-party JDBC drivers are installed with WebLogic Server:

- Oracle Thin Driver 10g version of the Oracle Thin driver (`ojdbc14.jar`)
- Sybase jConnect 4.5 (`jConnect.jar`), 5.5 (`jconn2.jar`), and 6.0 (jconn3.jar)
- MySQL 5.0.x (`mysql-connector-java-commercial-5.0.x-bin.jar`)

These drivers are installed in the *WL_HOME*\server\lib folder (where *WL_HOME* is the folder where WebLogic Server is installed) with weblogic.jar. The manifest in weblogic.jar lists these files so that they are loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add these JDBC drivers to your CLASSPATH. If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you must install the drivers, which includes updating your CLASSPATH with the path to the driver files, and may include updating your PATH with the path to database client files. See "Supported Database Configurations" in *Supported Configurations for WebLogic Server 10.0.*

Note: The WebLogic Type 4 JDBC drivers from DataDirect are also installed with WebLogic Server. See *WebLogic Type 4 JDBC Drivers* for more information.

If you plan to use a different version any of the drivers installed with WebLogic Server, you can replace the driver file in *WL_HOME*\server\lib with an updated version of the file or add the new file to the front of your CLASSPATH.

Copies of the drivers installed with WebLogic Server and other supporting files are installed in *WL_HOME*\server\ext\jdbc\. There is a subdirectory in this folder for each DBMS. If you need to revert to the version of the driver installed with WebLogic Server, you can copy the file from *WL_HOME*\server\ext\jdbc*DBMS* to *WL_HOME*\server\lib.

Note: WebLogic Server also includes the PointBase 5.1 JDBC driver and an evaluation version of the PointBase DBMS installed with the WebLogic Server examples in *WL_HOME*\common\eval\pointbase. PointBase Server is an all-Java DBMS product included in the WebLogic Server distribution solely in support of WebLogic Server evaluation, either in the form of custom trial applications or through packaged sample applications provided with WebLogic Server. Non-evaluation development or production use of the PointBase Server requires a separate license be obtained by the end user directly from PointBase.

Setting the Environment for a Type-4 Third-Party JDBC Driver

If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you need to update the WebLogic Server's classpath to include the location of the JDBC driver classes. Edit the commEnv.cmd/sh script in *WL_HOME*/common/bin and prepend your classes as described in "Modifying the Classpath" in *WebLogic Server Command Reference*.

Globalization Support for the Oracle 10g Thin Driver

For Globalization Support with the 10g version of the Oracle Thin driver, Oracle supplies the `orai18n.jar` file, which replaces `nls_charset.zip`. If you use character sets other than US7ASCII, WE8DEC, WE8ISO8859P1 and UTF8 with CHAR and NCHAR data in Oracle object types and collections, you must include `orai18n.jar` in your `CLASSPATH`. `orai18n.jar` is included with the WebLogic Server installation in the *`WL_HOME`*`\server\ext\jdbc\oracle\10g` folder. The file is *not* referenced by the weblogic.jar manifest file, so you must add it to your `CLASSPATH` before you can use it.

Using the Oracle Thin Driver in Debug Mode

The *`WL_HOME`*`\server\ext\jdbc\oracle\10g` folder includes the `ojdbc14_g.jar` file, which is the version of the Oracle Thin driver with classes to support debugging and tracing. To use the Oracle Thin driver in debug mode, add the path to this file at the beginning of your `CLASSPATH` .

CHAPTER 6

Monitoring WebLogic JDBC Resources

This release of WebLogic Server includes the WebLogic Diagnostic Service, which is a monitoring and diagnostic service that runs within the WebLogic Server process and participates in the standard server life cycle. This service enables you to create, collect, analyze, archive, and access diagnostic data generated by a running server and the applications deployed within its containers. This data provides insight into the runtime performance of servers and applications and enables you to isolate and diagnose faults when they occur. WebLogic JDBC takes advantage of this service to provide enhanced runtime statistics, profile information over a period of time, logging, and debugging to help you keep your WebLogic domain running smoothly.

You can use the runtime statistics to monitor the data sources in your WebLogic domain to see if there is a problem. If there is a problem, you can use profiling to determine which application is the source of the problem. Once you've narrowed it down to the application, you can then use JDBC debugging features to find the problem within the application.

The following sections include details about monitoring JDBC objects:

- Viewing Runtime Statistics
- Collecting Profile Information
- Debugging JDBC Data Sources

For more information about the WebLogic Diagnostic Service, see Understanding the WebLogic Diagnostic Service.

Viewing Runtime Statistics

Viewing runtime statistics allows you to monitor the data sources in your WebLogic domain.

Data Source Statistics

You can view runtime statistics for a data source via the Administration Console or through the `JBCDataSourceRuntimeMBean`. The `JDBCDataSourceRuntimeMBean` provides methods for getting the current state of the data source and for getting statistics about the data source, such as the average number of active connections, the current number of active connections, the highest number of active connections, and so forth. For more information, see JDBCDataSourceRuntimeMBean in the *WebLogic Server MBean Reference*.

Prepared Statement Cache Statistics

You can view runtime statistics for a prepared statement cache via the Administration Console or through the `JBCDataSourceRuntimeMBean`. For more information, see JDBCDataSourceRuntimeMBean in the *WebLogic Server MBean Reference*.

Collecting Profile Information

If the statistics that you are seeing indicate that there is a problem in your WebLogic Server domain, you can configure any data source to collect profile information to help you pinpoint the source of the problem. The collected profile information is stored in records in the WLDF Archive. The fields contain different information for different profile types, as described in the sections that follow.

When configuring your data source for profiling, you must specify the interval at which profile data is harvested (`Harvest Frequency Seconds`); if the interval is set to 0, harvesting of data is disabled. For more information, see Configuring the Harvester

Profile Types

You can choose to profile the following information about data sources and the prepared statement cache, as described in the next sections of this document:

- Connection Usage (PROFILE_TYPE_CONN_USAGE_STR)
- Connection Reservation Wait (PROFILE_TYPE_CONN_RESV_WAIT_STR)
- Connection Reservation Failed (PROFILE_TYPE_CONN_RESV_FAIL_STR)

- Connection Leak (PROFILE_TYPE_CONN_LEAK_STR)
- Connection Last Usage (PROFILE_TYPE_CONN_LAST_USAGE_STR)
- Connection Multithreaded Usage (PROFILE_TYPE_CONN_MT_USAGE_STR)
- Statement Cache Entry (PROFILE_TYPE_STMT_CACHE_ENTRY_STR)
- Statements Usage (PROFILE_TYPE_STMT_USAGE_STR)

Connection Usage (PROFILE_TYPE_CONN_USAGE_STR)

Enable connection usage profiling to collect information about threads currently using connections from the pool of connections in the data source. This profile information can help determine why applications are unable to get connections from the data source.

Note: By default, enabling connection usage profiling on its own will not provide a stack trace of the threads using the connections. To obtain this information you must enable profiling of connection leaks in addition to enabling connection. For more information about profiling connection leaks see "Connection Leak (PROFILE_TYPE_CONN_LEAK_STR)" on page 6-4.

The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - connection ID
- User - stack trace of the thread using the connection
- Timestamp - time stamp showing when the connection was given to the thread

Connection Reservation Wait (PROFILE_TYPE_CONN_RESV_WAIT_STR)

Enable connection reservation wait profiling to collect information about threads currently waiting to reserve a connection from the data source. This profile information can help determine why applications are unable to get connections from the data source or to wait for connections. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - thread ID
- User - stack trace of the thread waiting for the connection
- Timestamp - time stamp showing when the thread started waiting for a connection

Connection Reservation Failed (PROFILE_TYPE_CONN_RESV_FAIL_STR)

Enable connection reservation failure profiling to collect information about threads that attempt to reserve a connection from the data source but fail to get that connection. This profile information can help determine why applications are unable to get connections from the data source even after reserving them. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - thread ID
- User - stack trace of the thread waiting for the connection plus the exception received when the reservation request failed
- Timestamp - time stamp showing when the reservation request failed

Connection Leak (PROFILE_TYPE_CONN_LEAK_STR)

Enable connection leak profiling to collect information about threads that have reserved a connection from the data source and the connection leaked (was not properly returned to the pool of connections). This profile information can help determine which applications are not properly closing JDBC connections. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - connection ID
- User - stack trace of the thread waiting for the connection
- Timestamp - time stamp showing when the connection leak was detected

Connection Last Usage (PROFILE_TYPE_CONN_LAST_USAGE_STR)

Enable connection last usage profiling to collect information about the previous thread that last used the connection. This information is useful when you are debugging problems with connections infected in pending transactions that cause subsequent XA operations on the connections to fail. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - stack trace of the XA exception thrown
- User - stack trace of the thread that last used the connection
- Timestamp - timestamp showing when the exception was thrown

Connection Multithreaded Usage (PROFILE_TYPE_CONN_MT_USAGE_STR)

Enable connection multithreaded usage profiling to collect information about threads that erroneously use a connection that was previously obtained by a different thread. This information is useful when an application reports a problem that you suspect may have been caused by the simultaneous use of a connection by more than one thread. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - stack trace of the other thread that was found using the connection
- User - stack trace of the thread that reserved the connection
- Timestamp - time stamp showing when usage of the connection by multiple threads was detected

Statement Cache Entry (PROFILE_TYPE_STMT_CACHE_ENTRY_STR)

Enable statement cache entry profiling to collect information for prepared and callable statements added to the statement cache, and for the threads that originated the cached statements. This information can help you determine how the cache is being used. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - string representation of the statement
- User - stack trace of the thread using the statement
- Timestamp - time stamp showing when the statement was added to the cache

Statements Usage (PROFILE_TYPE_STMT_USAGE_STR)

Enable statements usage profiling to collect information about threads currently executing SQL statements from the statement cache. This information can help you determine how statements are being used. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - SQL statement being executed via the statement
- User - stack trace of the thread using the statement
- Timestamp - duration of statement execution

Accessing Diagnostic Data

Once profile data has been collected, you can access it using code similar to that shown in the sample below, substituting the appropriate profile type for the one shown in boldface type. The first part of the code defines a vector variable (`dataV`) in which the collected profile data (`jdbcProfData`) is stored. The second part of the code gets the stored data from the vector variable and prints it.

```
}
jdbcProfData = getData("EventsDataArchive", "TYPE LIKE '%" +
JDBCLegalHelper.PROFILE_TYPE_CONN_USAGE_STR + "%'");
Vector dataV = new Vector();
for (int i = 0; i < jdbcProfData.length; i++) {
       if (jdbcProfData[i].getPoolName().equalsIgnoreCase
                     (Data_Source_1))
              dataV.add(jdbcProfData[i]);
}

System.out.println("records found for PROFILE_TYPE_CONN_USAGE_STR : " +
                     dataV.size());
for (int a = 0; a < dataV.size(); a++) {
       System.out.println("ID : " + ((ProfileDataRecord)
                     dataV.get(a)).getId());
       System.out.println("PoolName : " + ((ProfileDataRecord)
                     dataV.get(a)).getPoolName());
       System.out.println("Time : " + ((ProfileDataRecord)
                     dataV.get(a)).getTimestamp());
       System.out.println("User : " + ((ProfileDataRecord)
                     dataV.get(a)).getUser());
}
```

Another method for accessing diagnostic data is to use the Data Accessor component of the WebLogic Diagnostic Framework (WLDF); for more information see "Accessing Diagnostic Data Using the Data Accessor" in *Configuring and Using the WebLogic Diagnostic Framework.*

Callbacks for Monitoring Driver-Level Statistics

WebLogic Server provides callbacks for methods called on a JDBC driver. You can use these callbacks to monitor and profile JDBC driver usage, including methods being executed, any exceptions thrown, and the time spent executing driver methods.

To enable the callback feature, you specify the fully qualified path of the callback handler for the driver-interceptor element in the JDBC data source descriptor (module). Your callback handler must implement the weblogic.jdbc.extensions.DriverInterceptor interface. When you enable JDBC driver callbacks, WebLogic Server calls the `preInvokeCallback()`, `postInvokeExceptionCallback()`, and `postInvokeCallback()` methods of the registered callback handler before and after invoking any method inside the JDBC driver.

Any time an application calls the JDBC driver, a callback is sent to the class that implemented the driver.

Debugging JDBC Data Sources

Once you have narrowed the problem down to a specific application, you can activate WebLogic Server's debugging features to track down the specific problem within the application.

Enabling Debugging

You can enable debugging by setting the appropriate `ServerDebug` configuration attribute to "`true`." Optionally, you can also set the server `StdoutSeverity` to "`Debug`".

You can modify the configuration attribute in any of the following ways.

Enable Debugging Using the Command Line

Set the appropriate properties on the command line. For example,

```
-Dweblogic.debug.DebugJDBCSQL=true
-Dweblogic.log.StdoutSeverity="Debug"
```

This method is static and can only be used at server startup.

Enable Debugging Using the WebLogic Server Administration Console

Use the WebLogic Server Administration Console to set the debugging values:

1. If you have not already done so, in the Change Center of the Administration Console, click Lock & Edit (see Use the Change Center).

2. In the left pane of the console, expand Environment and select Servers.
3. On the Summary of Servers page, click the server on which you want to enable or disable debugging to open the settings page for that server.
4. Click Debug.
5. Expand default.
6. Select the check box for the debug scopes or attributes you want to modify.
7. Select Enable to enable (or Disable to disable) the debug scopes or attributes you have checked.
8. To activate these changes, in the Change Center of the Administration Console, click Activate Changes.
   Not all changes take effect immediately—some require a restart (see Use the Change Center).

   This method is dynamic and can be used to enable debugging while the server is running.

Enable Debugging Using the WebLogic Scripting Tool

Use the WebLogic Scripting Tool (WLST) to set the debugging values. For example, the following command runs a program for setting debugging values called `debug.py`:

```
java weblogic.WLST debug.py
```

The debug.py program contains the following code:

```
user='user1'
password='password'
url='t3://localhost:7001'
connect(user, password, url)
edit()
cd('Servers/myserver/ServerDebug/myserver')
startEdit()
set('DebugJDBCSQL','true')
save()
activate()
```

Note that you can also use WLST from Java. The following example shows a Java file used to set debugging values:

```
import weblogic.management.scripting.utils.WLSTInterpreter;
import java.io.*;
```

```
import weblogic.jndi.Environment;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class test {
        public static void main(String args[]) {
       try {
              WLSTInterpreter interpreter = null;
              String user="user1";
              String pass="pw12ab";
              String url ="t3://localhost:7001";
              Environment env = new Environment();
              env.setProviderUrl(url);
              env.setSecurityPrincipal(user);
              env.setSecurityCredentials(pass);
              Context ctx = env.getInitialContext();

              interpreter = new WLSTInterpreter();
              interpreter.exec
                     ("connect('"+user+"','"+pass+"','"+url+"')");
              interpreter.exec("edit()");
              interpreter.exec("startEdit()");
              interpreter.exec
                     ("cd('Servers/myserver/ServerDebug/myserver')");
              interpreter.exec("set('DebugJDBCSQL','true')");
              interpreter.exec("save()");
              interpreter.exec("activate()");

       } catch (Exception e) {
       System.out.println("Exception "+e);
       }
       }
}
```

Using the WLST is a dynamic method and can be used to enable debugging while the server is running.

Changes to the config.xml File

Changes in debugging characteristics, through console, or WLST, or command line are persisted in the `config.xml` file. See Figure 6-1, "Example Debugging Stanza for JDBC," on page 6-10:

Listing 6-1 Example Debugging Stanza for JDBC

```
.
.
.
<server>
<name>myserver</name>
<server-debug>
<debug-scope>
<name>weblogic.transaction</name>
<enabled>true</enabled>
</debug-scope>
<debug-jdbcsql>true</debug-jdbcsql>
</server-debug>
</server>
.
.
.
```

This sample `config.xml` fragment shows a transaction debug scope (set of debug attributes) and a single JDBC attribute.

JDBC Debugging Scopes

It is possible to see the tree view of the DebugScope definitions using `java weblogic.diagnostics.debug.DebugScopeViewer`.

You can enable the following registered debugging scopes for JDBC:

- DebugJDBCSQL (scope weblogic.jdbc.sql) - prints information about all JDBC methods invoked, including their arguments and return values, and thrown exceptions.
- DebugJDBCConn (scope weblogic.jdbc.connection) - trace all connection reserve and release operations in data sources as well as all application requests to get or close connections.
- DebugJDBCRMI (scope weblogic.jdbc.rmi) - similar to JDBCSQL but at the RMI level; turning on this one and JDBCSQL will get two sets of debug messages for each operation called from a client.
- DebugJDBCInternal (scope weblogic.jdbc.internal) - low level debugging in weblogic/jdbc/common/internal related to the data source, the connection environment, and the data source manager.
- DebugJDBCDriverLogging (scope weblogic.jdbc.driverlogging) - enables JDBC driver level logging (this replaces ServerMBean JDBCLoggingEnabled and getJDBCLogFileName). Note that to get driver level tracing for Oracle, you need to use ojdbc14_g.jar instead of ojdbc14.jar. Note that for this debug scope, it can be turned on once via the command line or configuration when the server is booted but cannot be turned on or off dynamically (due to the DriverManager interface).

  Note: BEA WebLogic JDBC Spy logs detailed information about JDBC calls issued by an application and then passes the calls to the wrapped WebLogic Type 4 JDBC driver. You can use the information in the logs to help troubleshoot problems in your application. For more information about WebLogic JDBC Spy, see "Tracking JDBC Calls with WebLogic JDBC Spy" in *WebLogic Type 4 JDBC Drivers.*

Request Dyeing

Another option for debugging is to trace the flow of an individual (typically "dyed") application request through the JDBC subsystem. For more information, see "Configuring the Dye Vector via the DyeInjection Monitor" in *Configuring and Using the WebLogic Diagnostic Framework.*

CHAPTER 7

Managing WebLogic JDBC Resources

Using the Administration Console, JMX programs, or WebLogic Scripting Tool (WLST) scripts, you can manage the JDBC data sources in your domain. This section includes the following information:

- "Testing Data Sources and Database Connections" on page 7-1
- "Managing the Statement Cache for a Data Source" on page 7-2
- "Shrinking a Data Source" on page 7-3
- "Resetting a Data Source" on page 7-4
- "Suspending a Data Source" on page 7-4
- "Resuming a Data Source" on page 7-4
- "Shutting Down a Data Source" on page 7-5
- "Starting a Data Source" on page 7-5

Testing Data Sources and Database Connections

`JDBCDataSourceRuntimeMBean.testPool`

To make sure that the database connections in a data source remain healthy, you should periodically test the connections. WebLogic Server includes two basic types of testing: automatic testing that you configure with attributes on the data source and manual testing that you can do to trouble-shoot a data source.

Allowing the WebLogic Server to automatically maintain the integrity of pool connections should prevent most DBMS connection problems. For more information about configuring automatic connection testing, see "Connection Testing Options for a Data Source" on page 3-23.

To manually test a connection from a data source, you can use the Test Data Source feature on the JDBC Data Source: Monitoring: Testing page in the Administration Console (see "Test JDBC data sources") or the `testPool()` method in the `JDBCDataSourceRuntimMBean`. To test a database connection from a data source, Test Reserved Connections must be enabled and Test Table Name must be defined in the data source configuration. Both are defined by default if you create the data source using the Administration Console.

When you test a data source, WebLogic Server reserves a connection, tests it using the query defined in Test Table Name, and then releases the connection.

Managing the Statement Cache for a Data Source

For each connection in a data source in your system, WebLogic Server creates a statement cache. When a prepared statement or callable statement is used on a connection, WebLogic Server caches the statement so that it can be reused. For more information about the statement cache, see "Increasing Performance with the Statement Cache" on page 3-20.

Each connection in the data source has its own statement cache, but configuration settings are made for all connections in the data source. You can clear the statement cache for *all* connections in a data source using the Administration Console or you can programmatically clear the statement cache for an *individual* connection.

Clearing the Statement Cache for a Data Source

`JDBCDataSourceRuntimeMBean.clearStatementCache`

You can manually clear the statement cache for all connections in a data source using the Administration Console (see "Clear the statement cache in a JDBC data source") or with the `clearStatementCache()` method on the `JDBCDataSourceRuntimeMBean`.

Clearing the Statement Cache for a Single Connection

```
weblogic.jdbc.extensions.WLConnection.clearStatementCache()
weblogic.jdbc.extensions.WLConnection.clearCallableStatement(java.lang.
String sql)
weblogic.jdbc.extensions.WLConnection.clearCallableStatement(java.lang.
String sql,int resType,int resConcurrency)
```

```
weblogic.jdbc.extensions.WLConnection.clearPreparedStatement(java.lang.
String sql)
weblogic.jdbc.extensions.WLConnection.clearPreparedStatement(java.lang.
String sql,int resType,int resConcurrency)
```

You can use methods in the `weblogic.jdbc.extensions.WLConnection` interface to clear the statement cache for a single connection or to clear an individual statement from the cache. These methods return `true` if the operation was successful and `false` if the operation fails because the statement was not found.

When prepared and callable statements are stored in the cache, they are stored (keyed) based on the exact SQL statement and result set parameters (type and concurrency options), if any. When clearing an individual prepared or callable statement, you must use the method that takes the proper result set parameters. For example, if you have callable statement in the cache with `resSetType` of `ResultSet.TYPE_SCROLL_INSENSITIVE` and a `resSetConcurrency` of `ResultSet.CONCUR_READ_ONLY`, you must use the method that takes the result set parameters:

```
clearCallableStatement(java.lang.String sql,int resSetType,int
resSetConcurrency)
```

If you use the method that only takes the SQL string as a parameter, the method will not find the statement, nothing will be cleared from the cache, and the method will return `false`.

When you clear a statement that is currently in use by an application, WebLogic Server removes the statement from the cache, but does not close it. When you clear a statement that is not currently in use, WebLogic Server removes the statement from the cache and closes it.

For more details about these methods, see the Javadoc for WLConnection.

Shrinking a Data Source

`JDBCDataSourceRuntimeMBean.shrink`

A data source has a set of properties that define the initial and maximum number of connections in the pool (`initialCapacity` and `maxCapacity`), and the number of connections to add to the pool when all connections are in use (`capacityIncrement`). When the pool reaches its maximum capacity, the maximum number of connections are opened, and they remain opened unless you enable automatic shrinking on the data source or manually shrink the data source with the `shrink()` method.

You may want to drop some connections from the data source when a peak usage period has ended, freeing up WebLogic Server and DBMS resources. You can use the Shrink option on the JDBC Data Source: Control page in the Administration Console (see "Shrink the connection pool

in a JDBC data source") or the `shrink()` method on the `JDBCDataSourceRuntimeMBean`. When you shrink a data source, WebLogic Server reduces the number of connections in the pool to the greater of either the initial capacity or the number of connections currently in use.

Resetting a Data Source

`JDBCDataSourceRuntimeMBean.reset`

To close and recreate all available database connections in a data source, you can use the Reset option on the JDBC Data Source: Control page in the Administration Console (see "Reset connections in a JDBC data source") or the `reset()` method on the `JDBCDataSourceRuntimeMBean`. This may be necessary after the DBMS has been restarted, for example. Often when one connection in a data source has failed, all of the connections in the pool are bad.

Suspending a Data Source

`JDBCDataSourceRuntimeMBean.suspend`

`JDBCDataSourceRuntimeMBean.forceSuspend`

To suspend a data source, you can use the Suspend and Force Suspend options on the JDBC Data Source: Control page in the Administration Console (see "Suspend JDBC data sources") or the `suspend()` and `forceSuspend()` methods in the `JDBCDataSourceRuntimeMBean`.

When you suspend a data source (not forcibly suspend), the data source is marked as disabled and applications cannot use connections from the pool. Applications that already have a reserved connection from the data source when it is suspended will get an exception when trying to use the connection. WebLogic Server preserves all connections in the data source exactly as they were before the data source was suspended.

When you forcibly suspend a data source, WebLogic Server marks the data source as disabled, forcibly disconnects applications that are currently using a connection, and recreates (closes and reopens) connections that were in use when the data source was suspended. Any transaction on the connections that are closed are rolled back. WebLogic Server preserves all other connections exactly as they were before the data source was suspended.

Resuming a Data Source

`JDBCDataSourceRuntimeMBean.resume`

To re-enable a data source that you suspended, you can use the Resume option on the JDBC Data Source: Control page in the Administration Console (see "Resume suspended JDBC data sources") or the `resume()` method on the `JDBCDataSourceRuntimeMBean`. When you resume a data source, WebLogic Server marks the data source as enabled and allows applications to use connections from the data source. If you suspended the data source (not forcibly suspended), all connections are preserved exactly as they were before the data source was suspended. Clients that had reserved a connection before the data source was suspended can continue JDBC operations exactly where they left off. If you forcibly suspended the data source, connections that were not in use when the data source was suspended are preserved exactly as they were before the suspension. Connections that *were* in use were closed and reopened. Clients that had reserved a connection no longer have a valid JDBC context.

Note: You cannot resume a data source that did not start correctly, for example, if the database server is unavailable.

Shutting Down a Data Source

`JDBCDataSourceRuntimeMBean.shutdown`

`JDBCDataSourceRuntimeMBean.forceShutdown`

To shut down a data source, you can use the Shutdown and Force Shutdown options on the JDBC Data Source: Control page in the Administration Console (see "Shut down JDBC data sources") or the `shutdown()` and `forceShutdown()` methods in the `JDBCDataSourceRuntimeMBean`.

When you shut down a data source (not forcibly shut down), WebLogic Server closes database connections in the data source and shuts down the data source. If any connections from the data source are currently in use, the operation will fail.

When you forcibly shut down a data source, WebLogic Server closes database connections in the data source and shuts down the data source. All current connection users are forcibly disconnected.

Starting a Data Source

`JDBCDataSourceRuntimeMBean.start`

After you shut down a data source, you can use the Start option on the JDBC Data Source: Control page in the Administration Console (see "Start JDBC data sources") or the `start()` method in the `JDBCDataSourceRuntimeMBean`. Invoking the Start operation re-initializes the data source, creates connections and transitions the data source to a health state of Running.

APPENDIX A

Configuring JDBC Application Modules for Deployment

When you package your enterprise application, you can include JDBC resources in the application by packaging JDBC modules in the archive and adding references to the JDBC modules in all applicable descriptor files. When you deploy the application, the JDBC resources are deployed, too. Depending on how you configure the JDBC modules, the JDBC data sources deployed with the application will either be restricted for use only by the containing application (*application-scoped modules*) or will be available to all applications and clients (*globally-scoped modules*).

The following sections describe the details of packaged JDBC modules:

- "Packaging a JDBC Module with an Enterprise Application: Main Steps" on page A-1
- "Creating Packaged JDBC Modules" on page A-2
- "Referencing a JDBC Module in Java EE Descriptor Files" on page A-7
- "Packaging an Enterprise Application with a JDBC Module" on page A-10
- "Deploying an Enterprise Application with a JDBC Module" on page A-10
- "Getting a Database Connection from a Packaged JDBC Module" on page A-10

Packaging a JDBC Module with an Enterprise Application: Main Steps

The main steps for creating, packaging, and deploying a JDBC module with an enterprise application are as follows:

1. Create the module. See "Creating Packaged JDBC Modules" on page A-2.
2. Add references to the module in all applicable descriptor files. See "Referencing a JDBC Module in Java EE Descriptor Files" on page A-7.
3. Package all application modules in an EAR. See "Packaging an Enterprise Application with a JDBC Module" on page A-10.
4. Deploy the application. See "Deploying an Enterprise Application with a JDBC Module" on page A-10.

Creating Packaged JDBC Modules

You can create JDBC application modules using any development tool that supports creating an XML descriptor file. You then deploy and manage JDBC modules using JSR 88-based tools, such as the `weblogic.Deployer` utility, or the Administration Console.

Note: You can create a JDBC data source using the Administration Console, then copy the module as a template for use in your applications. You must change the `name` and `jndi-name` elements of the module before deploying it with your application to avoid a naming conflict in the namespace.

Each JDBC module represents a data source or a multi data source. Modules that represent a data source include all of the configuration parameters for the data source. Modules that represent a multi data source include configuration parameters for the multi data source, including a list of data source modules used by the multi data source.

Creating a JDBC Data Source Module Using the Administration Console

To create a data source module in the Administration Console that you can re-use as an application module, follow these steps.

1. Create a data source as described in "Creating a JDBC Data Source" on page 3-2. The data source module is created in the `config/jdbc` subdirectory of the domain directory.
2. Copy the `data-source-name.xml` file to a subdirectory within your application and rename the copy to include -jdbc as a suffix, such as `new-data-source-name-jdbc.xml`.
3. Open the file in an editor and change the following elements:
   - `name`—change the `name` to a name that is unique within the domain.

  - `jndi-name`—change the `jndi-name` to a name that you want the enterprise application to use to lookup the data source in the local application context.
  - `scope`—optionally, to limit access to the data source to only the containing application, add a `scope` element to the `jdbc-data-source-params` section of the module. For example, `<scope>Application</scope>`. See "Application Scoping for a Packaged JDBC Module" on page A-7.

4. Continue with adding references to the descriptor files in the enterprise application. See "Referencing a JDBC Module in Java EE Descriptor Files" on page A-7.

JDBC Packaged Module Requirements

A JDBC module must meet the following criteria:

- Conforms to the `weblogic-jdbc.xsd` schema. The schema is available at `http://www.bea.com/ns/weblogic/920/weblogic-jdbc.xsd`.
- Uses a file name that ends in `-jdbc.xml`.
- Includes a `name` element that is unique within the WebLogic domain.

Data source modules must also include the following JDBC driver parameters:

- `url`
- `driver-name`
- `properties`, including any properties required by the JDBC driver to create database connections, such as a user name and password.

Multi data source modules must also include the following data source parameters:

- `data-source-list`, which is a list of data source modules, separated by commas, that the multi data source uses to satisfy database connection requests from applications.

Note: All data sources listed in the data-source-list must have the same XA and transaction protocol settings.

All other configuration parameters are optional or have a default value that WebLogic Server uses if a value is not specified. However, to create a useful JDBC module, you will likely need to specify additional configuration options as required by your applications and your environment.

JDBC Application Module Limitations

Note the following limitations for JDBC application modules:

- The LoggingLastResource global-transactions-protocol is not permitted for use in JDBC application modules.
- When deploying an application in production with application-scoped JDBC resources, if the resource uses EmulateTwoPhaseCommit for the global-transactions-protocol, you cannot deploy multiple versions of the application at the same time.

Creating a JDBC Data Source Module

The main sections within a JDBC data source module are:

- `jdbc-driver-params`—includes entries for the JDBC driver used to create database connections, including `url`, `driver-name`, and individual driver `property` entries. See the `weblogic-jdbc.xsd` schema for more valid entries. For an explanation of each element, see "JDBCDriverParamsBean" in the *WebLogic Server MBean Reference*.
- `jdbc-connection-pool-params`—includes entries for connection pool configuration, including connection testing options, statement cache options, and so forth. This element also inherits `connection-pool-params` from the `weblogic-j2ee.xsd` schema, including `initial-capacity`, `max-capacity`, and other options common to pooled resources. For more information, see the following:
  - "JDBCConnectionPoolParamsBean" in the *WebLogic Server MBean Reference*
  - `weblogic-jdbc.xsd` schema
- `jdbc-data-source-params`—includes entries for data source behavior options and transaction processing options, such as `jndi-name`, `row-prefetch-size`, and `global-transactions-protocol`. See the `weblogic-jdbc.xsd` schema for more valid entries. For an explanation of each element, see "JDBCDataSourceParamsBean" in the *WebLogic Server MBean Reference*.
- `jdbc-xa-params`—includes entries for XA database connection handling options, such as `keep-xa-conn-till-tx-complete`, and `xa-transaction-timeout`. For an explanation of each element, see "JDBCXAParamsBean" in the *WebLogic Server MBean Reference*.

Listing A-1 shows an example of a JDBC module for a data source with some typical configuration options.

Listing A-1 Sample JDBC Data Source Module

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91">
  <name>examples-demoXA-2</name>
```

```
<jdbc-driver-params>
  <url>jdbc:pointbase:server://localhost:9092/demo</url>
  <driver-name>com.pointbase.xa.xaDataSource</driver-name>
  <properties>
    <property>
      <name>user</name>
      <value>examples</value>
    </property>
    <property>
      <name>databaseName</name>
      <value>jdbc:pointbase:server://localhost:9092/demo</value>
    </property>
  </properties>
  <password-encrypted>eNEVN9dk4dEDUEVqL1</password-encrypted>
</jdbc-driver-params>

<jdbc-connection-pool-params>
  <initial-capacity>3</initial-capacity>
  <max-capacity>10</max-capacity>
  <test-connections-on-reserve>true</test-connections-on-reserve>
  <test-table-name>SQL SELECT COUNT(*) FROM SYSTABLES</test-table-name>
</jdbc-connection-pool-params>

<jdbc-data-source-params>
  <global-transactions-protocol>TwoPhaseCommit</global-transactions-protocol>
  <jndi-name>examples-demoXA-2</jndi-name>
  <scope>Application</scope>
</jdbc-data-source-params>

</jdbc-data-source>
```

Creating a JDBC Multi Data Source Module

A JDBC multi data source module is much simpler than a data source module. Only one main section is required: `jdbc-data-source-params`. The `jdbc-data-source-params` element in a multi data source differs in that it contains options for multi data source behavior options instead of data source behavior options. Only the following parameters in the `jdbc-data-source-params` are valid for multi data sources:

- `jndi-name` (required)
- `data-source-list` (required)
- `scope`

- `algorithm-type`
- `connection-pool-failover-callback-handler`
- `failover-request-if-busy`

For an explanation of each element, see "JDBCDataSourceParamsBean" in the *WebLogic Server MBean Reference*.

Listing A-2 shows an example of a JDBC module for a data source with some typical configuration options.

Listing A-2 Sample JDBC Multi Data Source Module

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91">
  <name>examples-demoXA-multi-data-source</name>
  <jdbc-data-source-params>
    <jndi-name>examples-demoXA -multi-data-source</jndi-name>
    <algorithm-type>Load-Balancing</algorithm-type>
    <data-source-list>examples-demoXA,examples-demoXA-2</data-source-list>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Encrypting Database Passwords in a JDBC Module

BEA recommends that you encrypt database passwords in a JDBC module to keep your data secure. To encrypt a database password, you process the password with the WebLogic Server `encrypt` utility, which returns an encrypted equivalent of the password that you include in the JDBC module as the `password-encrypted` element. For more details about using the WebLogic Server encrypt utility, see "encrypt" in the *WebLogic Server Command Reference*.

Deploying JDBC Modules to New Domains

It it common practice for JDBC modules to be moved from one domain to another, such as moving an application from development to production. However, the encryption key generated by the WebLogic Server encrypt utility is not transferable to a new domain. When moving a JDBC module with an encrypted database password, you must do one of the following:

- Override the old encrypted password within a deployment plan that includes a password that was encrypted specifically for the new domain. See "Update a deployment plan" in *Administration Console Online Help*.
- Re-encrypt the passwords for your new domain. See "Encrypting Database Passwords in a JDBC Module" on page A-6

Application Scoping for a Packaged JDBC Module

By default, when you package a JDBC module with an application, the JDBC resource is globally scoped—that is, the resource is bound to the global JNDI namespace and is available to all applications and clients. To reserve the resource for use only by the enclosing application, you must include the `<scope>Application</scope>` parameter in the `jdbc-data-source-params` element in the JDBC module, which binds the resource to the local application namespace. For example:

```
<jdbc-data-source-params>
  <jndi-name>examples-demoXA-2</jndi-name>
  <scope>Application</scope>
</jdbc-data-source-params>
```

All data sources in a multi data source for an application-scoped JDBC module must also be application scoped.

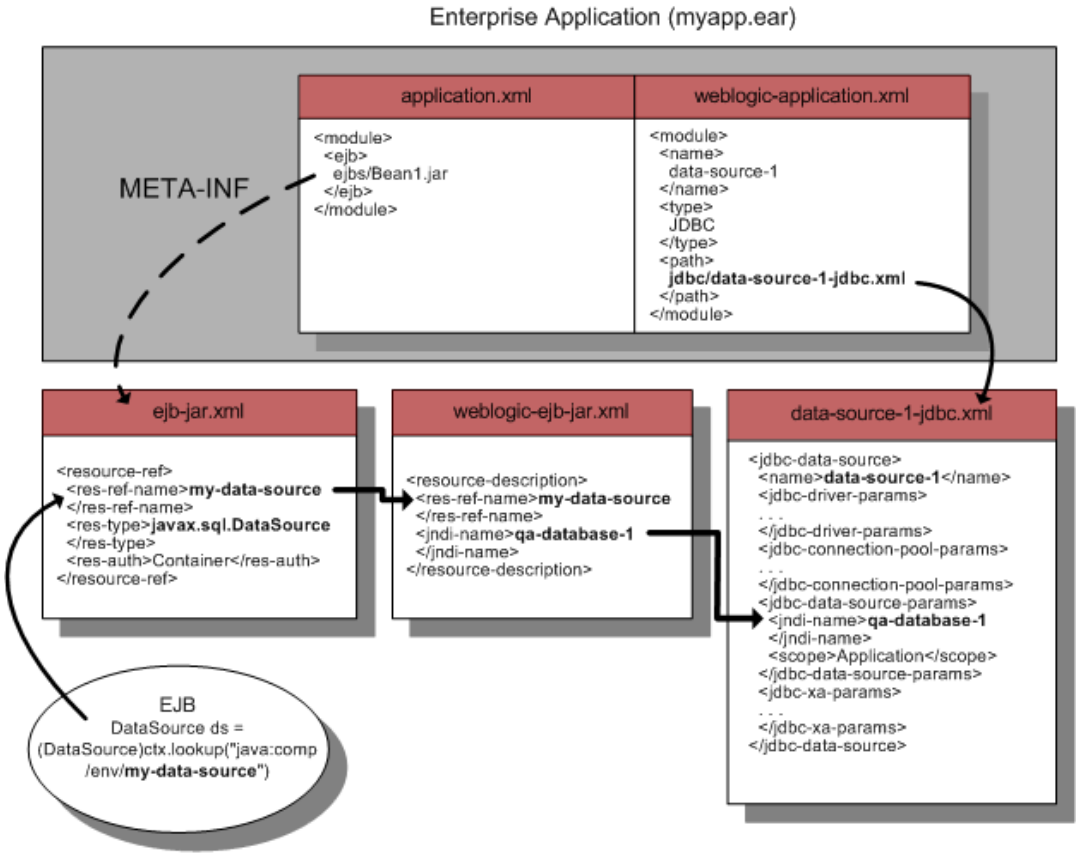
Referencing a JDBC Module in Java EE Descriptor Files

When you package a JDBC module with an enterprise application, you must reference the module in all applicable descriptor files, including among others:

- `weblogic-application.xml`
- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `web.xml`
- `weblogic.xml`

Figure A-1 shows the relationship between entries in various descriptor files for an EJB application and how they refer to a JDBC module packaged with the application.

Figure A-1 Relationship Between JDBC Modules and Descriptors in an Enterprise Application



Packaged JDBC Module References in weblogic-application.xml

When including JDBC modules in an enterprise application, you must list each JDBC module as a `module` element of type `JDBC` in the `weblogic-application.xml` descriptor file packaged with the application. For example:

```
<module>
  <name>data-source-1</name>
```

```
  <type>JDBC</type>
  <path>datasources/data-source-1-jdbc.xml</path>
</module>
```

Packaged JDBC Module References in Other Descriptors

For other application modules in your enterprise application to use the JDBC modules packaged with your application, you must add the following entries in the descriptor files packaged with application modules:

- In the standard Java EE descriptor files packaged with your application modules, such as `ejb-jar.xml` for an EJB, you must add `resource-ref-name` references to specify the JNDI name of the data source as used in the application. For example:

  ```
  <resource-ref>
    <res-ref-name>my-data-source</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  ```

  In this example, `my-data-source` is the data source name as used in the application module. Your application would look up the data source with the following code:

  ```
  javax.sql.DataSource ds =
      (javax.sql.DataSource) ctx.lookup("java:comp/env/my-data-source");
  ```

- In the WebLogic-specific descriptor files, such as weblogic-ejb-jar.xml for an EJB, you must map each `resource-ref-name` reference to the `jndi-name` element of a data source. For example:

  ```
  <resource-description>
    <res-ref-name>my-data-source</res-ref-name>
    <jndi-name>qa-database-1</jndi-name>
  </resource-description>
  ```

  In this example, the resource name (`<res-ref-name>my-data-source</res-ref-name>`) from the standard descriptor is mapped to the JNDI name (`<jndi-name>qa-database-1</jndi-name>`) of the data source in the JDBC module.

Figure A-1 shows the mapping of the of the data source name as used in the application module to the JNDI name of the JDBC data source in the JDBC module.

Note: For application-scoped data sources, if you do not add these entries to the descriptor files, your application will be unable to look up the data source to get a database connection.

Packaging an Enterprise Application with a JDBC Module

You package an application with a JDBC module as you would any other enterprise application. See "Packaging Applications Using wlpackage" in *Developing Applications with WebLogic Server*.

Deploying an Enterprise Application with a JDBC Module

You deploy an application with a JDBC module as you would any other enterprise application. See "Deploying Applications Using wldeploy" in *Developing Applications with WebLogic Server*.

Caution: When deploying an application in production with application-scoped JDBC resources, if the resource uses EmulateTwoPhaseCommit for the global-transactions-protocol, you cannot deploy multiple versions of the application at the same time.

Getting a Database Connection from a Packaged JDBC Module

To get a connection from JDBC module packaged with an enterprise application, you look up the data source or multi data source defined in the JDBC module in the local environment (`java:comp/env`) or on the JNDI tree and then request a connection from the data source or multi data source. For example:

```
javax.sql.DataSource ds =
    (javax.sql.DataSource) ctx.lookup("java:comp/env/my-data-source");
java.sql.Connection conn = ds.getConnection();
```

When you are finished using the connection, make sure you close the connection to return it to the connection pool in the data source:

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

APPENDIX B

Using WebLogic Server with Oracle RAC

More and more customers are looking for solutions to make their back-end systems more scalable and more available. In response to these requests, BEA supports Oracle Real Application Clusters (RAC) for use with WebLogic Server.

The following sections describe the requirements and configuration tasks for using Oracle Real Application Clusters with WebLogic Server:

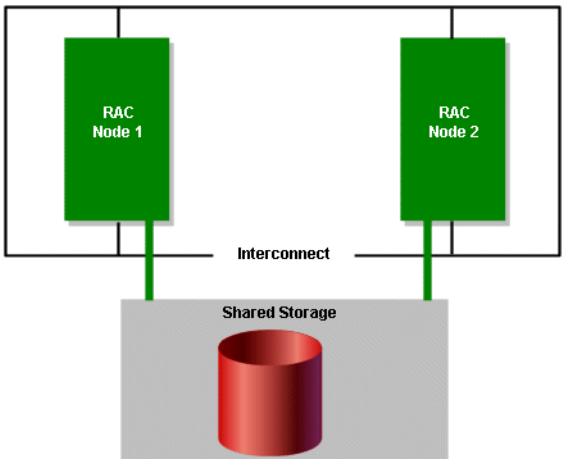
- Overview of Oracle Real Application Clusters
- Environment
- Configuration Considerations for Oracle
- Configuration Options in WebLogic Server with Oracle RAC
- XA Considerations and Limitations with Oracle RAC
- JDBC Store Recovery with Oracle RAC

Both Oracle RAC and WebLogic Server are complex systems. To use them together requires specific configuration on both systems, as well as clustering software and a shared storage solution. This document describes the configuration required at a high level. For more details about configuring Oracle RAC, your clustering software, your operating system, and your storage solution, see the documentation from the respective vendors.

Overview of Oracle Real Application Clusters

Oracle Real Application Clusters (RAC) is a software component you can add to a high-availability solution that enables users on multiple machines to access a single database with increased performance. RAC comprises two or more Oracle database instances running on two or more clustered machines and accessing a shared storage device via cluster technology. To support this architecture, the machines that host the database instances are linked by a high-speed interconnect to form the cluster. The interconnect is a physical network used as a means of communication between the nodes of the cluster. Cluster functionality is provided by the operating system or compatible third party clustering software. Figure B-1 shows an Oracle RAC configuration.

Figure B-1 Oracle Real Application Clusters Configuration



Oracle RAC offers features in the following areas:

- Scalability
- Availability
- Load balancing
- Failover

Oracle RAC Scalability with WebLogic Server

An Oracle RAC installation appears like a single standard Oracle database and is maintained using the same tools and practices. All the nodes in the cluster execute transactions against the same database and RAC coordinates each node's access to the shared data to maintain consistency and ensure integrity. You can add nodes to the cluster easily and there is no need to partition data when you add them. This means that you can horizontally scale the database tier as usage and demand grows by adding RAC nodes, storage, or both. You can then scale WebLogic Server by adding a data source that maps to the new node.

Oracle RAC Availability with WebLogic Server

Because every RAC node in the cluster has equal access and authority, the loss of a node may impact performance but does not result in downtime. Depending upon your configuration, when a RAC node fails, in-flight transactions are redirected to another node in the cluster either by WebLogic Server or by the Oracle Thin driver. Note that Oracle RAC does not provide failover for database connections; nor does WebLogic Server. But transactions are failed over in the sense that they are driven to completion, based on the time of the failure.

Oracle RAC Load Balancing with WebLogic Server

If your application requires load balancing across RAC nodes, BEA supports this capability through use of JDBC multi data sources with Oracle RAC nodes. The data sources that form a multi data source are accessed using a round-robin scheme. When switching connections, WebLogic Server selects a connection from the next data source in the order listed. For more information about using multi data sources with Oracle RAC, see “Using Multi Data Sources with Oracle RAC” on page B-13.

When multi data sources are not an option, load balancing when using XA is not supported. In a configuration without a multi data source, WebLogic Server relies on the connect-time failover feature provided by the Oracle Thin driver to work with Oracle RAC. As described in Oracle's Technical Note 235118.1, the Oracle Thin driver cannot guarantee that a transaction is initiated and concluded on the same Oracle RAC instance when the driver is configured for load balancing. As Oracle RAC requires that all database operations inside a global transaction be routed to the same Oracle instance, this known limitation means that you cannot use driver-level load balancing when using XA with Oracle RAC and therefore you cannot use a primary/primary RAC configuration.

Oracle RAC Failover with WebLogic Server

Although Oracle RAC offers JDBC connect-time failover features, for most configurations, BEA recommends using WebLogic JDBC multi data sources to handle failover instead. While connect-time failover does not provide the ability to pre-create connections to alternate RAC nodes, multi data sources have multiple connections available at all times to handle failover. For more information see "Using Multi Data Sources with Oracle RAC" on page B-13.

Note: Transparent Application Failover (TAF) requires the use of the Oracle OCI driver. Because BEA requires the use of the Oracle Thin driver, TAF is not supported.

Environment

When using WebLogic Server with Oracle RAC, consider the following requirements:

- Hardware Requirements
- Software Requirements

Note: See the *WebLogic Platform Supported Configurations* documentation at `{PLATFORM}/index.html` for the latest WebLogic Server hardware platform and operating system support, and for the Oracle RAC versions supported by WebLogic Server versions and service packs. See the Oracle documentation for hardware and software requirements for running the Oracle RAC software.

Hardware Requirements

A typical WebLogic Server/Oracle RAC system includes a WebLogic Server cluster, an Oracle RAC cluster, and hardware for shared storage.

WebLogic Server Cluster

The WebLogic Server cluster can be configured in many ways and with various hardware options. See *Using WebLogic Server Clusters* for more details about configuring a WebLogic Server cluster.

Oracle RAC Cluster

For the latest hardware requirements for Oracle RAC, see the Oracle RAC documentation. However, to use Oracle RAC with WebLogic Server, you must run Oracle RAC instances on robust, production-quality hardware. The Oracle RAC configuration must deliver database processing performance appropriate for reasonably-anticipated application load requirements.

Unusual database response delays can lead to unexpected behavior during database failover scenarios.

Shared Storage

In an Oracle RAC configuration, all data files, control files, and parameter files are shared for use by all RAC instances. An HA storage solution that uses one of the following architectures is recommended:

- Direct Attached Storage (DAS), such as a dual ported disk array or a Storage Area Network (SAN)
- Network Attached Storage (NAS)

For a complete list of supported storage solutions, see your Oracle documentation.

Software Requirements

To use WebLogic Server with Oracle RAC, you must install the following software on each RAC node:

- Operating system patches required to support Oracle RAC. See the release notes from Oracle for details.
- Oracle 9i or Oracle 10g database management system
- Clustering software for your operating system. See the Oracle documentation for supported clustering software and cluster configurations.
- Shared storage software, such as Veritas Cluster File System. Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

Note: See the *WebLogic Platform Supported Configurations* documentation at `{PLATFORM}/index.html` for the latest WebLogic Server hardware platform and operating system support, and for the Oracle RAC versions supported by WebLogic Server versions and service packs. See the Oracle documentation for hardware and software requirements required for running the Oracle RAC software.

Configuration Considerations for Oracle

Once you have installed and configured Oracle RAC, you must configure the listener process for each RAC instance as described in the sections that follow. For information about installing and

configuring your operating system and the Oracle software for Oracle RAC nodes see the Oracle documentation.

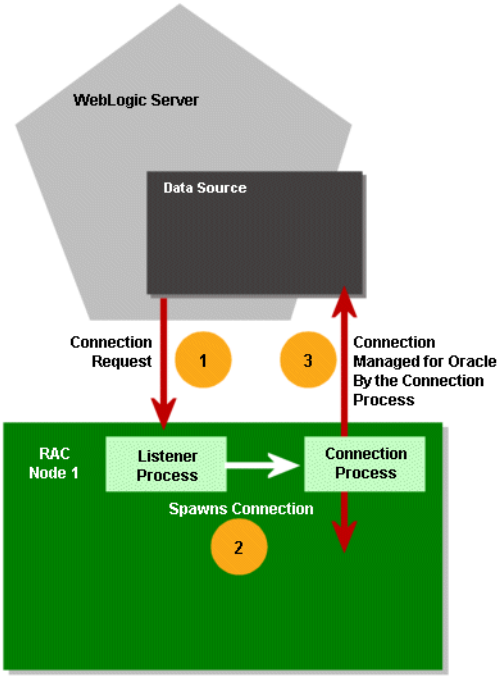
Configuring the Listener Process for Each Oracle RAC Instance

For Oracle RAC, the listener process establishes a communication pathway between a user process and an Oracle instance. When you use Oracle RAC with WebLogic Server, the user process is typically a JDBC data source.

When a data source is created, it attempts to create a pool of database connections for applications to borrow. If a pooled database connection becomes inoperative or if the data source is configured to grow in capacity, the data source attempts to create additional database connections up to the maximum specified in the configuration file. In all of these instances, the Oracle listener process handles the connection request on the Oracle RAC instance.

Figure B-2 shows the Oracle listener process functionality.

Figure B-2 Oracle Listener Process Functionality



To enable this functionality, you must configure the listener process for each RAC instance in the Oracle cluster. BEA requires that you configure a local listener on each RAC instance. Each database instance should be configured to register with its local listener only.

Oracle instances can be configured to register with the listener statically in the `listener.ora` file or registered dynamically using the instance initialization parameter `local_listener`, or both. BEA recommends using dynamic registration.

A listener can start either a shared dispatcher process or a dedicated process. When using with WebLogic Server, BEA recommends using dedicated processes.

Disabling Remote Listeners

Although Oracle RAC allows you to configure remote listeners to handle connection failover, remote listeners are typically too slow and BEA does not support their use. To disable remote listeners, delete any listed remote listeners in `spfile.ora` on each RAC node. For example:

```
*.remote_listener=''
```

BEA recommends using JDBC multi data sources to handle failover instead.

For more information see "Configuration Considerations for Failover" on page B-9.

Configuration Options in WebLogic Server with Oracle RAC

When using WebLogic Server with Oracle 9i RAC or Oracle 10g RAC, you must configure your WebLogic Domain so that it can interact with RAC instances and so that it performs as expected. The following sections describe configuration options and requirements:

- Choosing a WebLogic Server Configuration for Use with Oracle RAC
- Required JDBC Drivers
- Configuration Considerations for Failover
- Using Multi Data Sources with Oracle RAC
- Using Multi Data Sources with Global Transactions
- Using Multi Data Sources without Global Transactions
- Using Connect-Time Failover with Oracle RAC
- Using Connect-Time Failover without Global Transactions
- Using Fast Connection Failover

Choosing a WebLogic Server Configuration for Use with Oracle RAC

BEA supports several configuration options for using Oracle RAC with WebLogic Server:

- To connect to multiple Oracle 9i RAC or Oracle 10g RAC instances when using global transactions (XA), BEA recommends the use of transaction-aware WebLogic JDBC multi data sources, which support failover and load balancing, to connect to the RAC nodes. For more information see "Using Multi Data Sources with Global Transactions" on page B-14.
- To connect to multiple Oracle 9i RAC or Oracle10g RAC instances when not using XA, BEA recommends the use of (non-transaction-aware) multi data sources to connect to the RAC nodes. Use the standard multi data source configuration, which supports failover and

load balancing. For more information see "Using Multi Data Sources without Global Transactions" on page B-18.

- To connect to multiple Oracle RAC nodes when multi data sources are not an option and when not using XA, use Oracle 9i RAC or Oracle 10g RAC with connect-time failover. Note that load balancing is not supported in this configuration. For more information see "Using Connect-Time Failover without Global Transactions" on page B-23.

The following table may help you as you try to determine which configuration is right for your particular application:

Table B-1 Choosing Configurations to Use with Oracle Rac

Does Your Application Require				
Load Balancing?	**Failover?**	**Global Transactions (XA)?**	**JDBC Store?**	
Y	Y	Y	N	See "Using Multi Data Sources with Global Transactions" on page B-14
Y	Y	N	Y	See "Using Multi Data Sources without Global Transactions" on page B-18
N	Y	N	Y	See "Using Connect-Time Failover without Global Transactions" on page B-23

Required JDBC Drivers

To use WebLogic Server with Oracle RAC, your WebLogic JDBC data sources must use the Oracle JDBC Thin driver 10g to create database connections.

Configuration Considerations for Failover

Consider the following information when configuring for failover.

Multi Data Source-Managed Failover

Multi data sources offer failover for global transactions without the limitations and known issues associated with a data source configuration with connect-time failover. For a description of multi

data source failover features, see "Multi Data Source Failover Enhancements" in *Programming WebLogic JDBC*.

With this configuration, pictured in Figure B-3, you get:

- Faster failover controlled by the multi data source
- Automatic failback by the WebLogic Server health monitor

The multi data source handles failover for database connections when a RAC node becomes unavailable. When WebLogic Server tests a connection and the connection fails, it attempts to recreate the connection. If that attempt fails, the server disables the data source and routes connection requests to other data sources (which correspond to other RAC nodes) in the multi data source. WebLogic Server periodically tries to recreate the database connections in the disabled data source. When WebLogic Server is successful in recreating the connections, it next re-enables the data source and begins routing connection requests to the data source again. Because of the connection request routing and automatic health checking features, there is minimal delay in satisfying connection requests after a failure compared to when relying on the Oracle Thin driver connect-time failover configuration.

Connect-Time Failover

When multi data sources are not an option, WebLogic Server relies on the connect-time failover feature of the Oracle Thin driver to handle connection failover when a RAC instance becomes unavailable and a primary/primary configuration is not an option. When WebLogic Server tests a connection and the connection fails, the server replaces it by getting a new connection, and the driver again determines which RAC instance to use based on instance availability.When a connection fails a connection test, WebLogic Server automatically closes all connections in the data source. WebLogic Server replaces the connection with a new one, relying on the driver to determine to which node it should connect. In this case, the primary RAC node has failed, so the new connection is to the secondary RAC node. WebLogic Server tests the new connection before satisfying the request.

Delays During Failover

Occasionally, when one RAC node fails over to another, there may be a delay before the data associated with a transaction branch in progress on the now failed node is available throughout the cluster. This prevents incomplete transactions from being properly completed, which could further result in data locking in the database. To protect against the potential consequences of such a delay, WebLogic Server provides two configuration attributes that enable XA call retry for Oracle RAC: `XARetryDurationSeconds` and `XARetryIntervalSeconds`.

XARetryDurationSeconds controls the period of time during which WebLogic Server will repeatedly retry XA operations such as recover, commit and rollback for pending transactions. XARetryIntervalSeconds controls the frequency of the retry attempts within the established time period.

To enable XA call retries, add a value for XARetryDurationSeconds to all JDBC data sources in your WebLogic domain that connect to an Oracle RAC instance. For example:

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>oracleRACXAPool</name>
  ...
  <jdbc-xa-params>
    ...
    <xa-retry-duration-seconds>300</xa-retry-duration-seconds>
  </jdbc-xa-params>
</jdbc-data-source>
```

Use the following formula to determine the value for XARetryDurationSeconds:

> XARetryDurationSeconds = (longest transaction timeout for transactions that use connections from the data source) + (delay before XIDs are available on all RAC nodes, typically less than 5 minutes)

For example, if your application sets the longest transaction timeout as 180 seconds, you should set XARetryDurationSeconds to 180 seconds + 300 seconds, for a total of 480 seconds.

Note: It is generally better to set XARetryDurationSeconds higher than minimally necessary to make sure that all transactions are completed properly. Setting the value higher than minimally required should not affect application performance during normal operations. The additional processing only affects transactions that have been prepared but have failed to complete.

You can also optionally set a value for XARetryIntervalSeconds. This value determines the time between XA retry calls. By default, the value is 60 seconds. Decreasing the value will decrease the amount of time between XA retry attempts. The default value should suffice in most cases.

To enable XARetryDurationSeconds and XARetryIntervalSeconds from the Administration Console, use the following steps:

1. If you have not already done so, in the Change Center of the 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: Connection Pool** tab.
5. Scroll down and click **Advanced** to show the advanced connection pool options.
6. Update `XA Retry Duration` and `XA Retry Interval`.
7. Click **Save**.

Optionally, you can use the `weblogic.Admin` command line utility, the WebLogic Scripting Tool (WLST), or a JMX program.

Failure Handling Walkthrough for Global Transactions

What happens to inflight transactions to a database node if that node fails? When the primary Oracle RAC node fails? Does WebLogic Server support transparent failover? To answer these and other questions about how WebLogic Server handles failures, let's walk through the transaction processing steps and describe how a failure would be handled at each stage along the way.

The first stage at which a failure may occur is before the application calls for the transaction to be committed. If a database or RAC node fails at this stage, the application receives an exception and must get a new connection and make a new attempt at processing the transaction. WebLogic Server does not support transparent failover.

If a failure occurs after the application has called for the transaction to be committed, the handling of any in-flight transaction depends upon whether the `PREPARE` operation is complete. If the `PREPARE` operation is not complete, the transaction manager rolls back the transaction and sends the application an exception for the failed transaction. If the `PREPARE` operation is complete, the transaction manager attempts to drive the in-flight transaction to completion using another node.

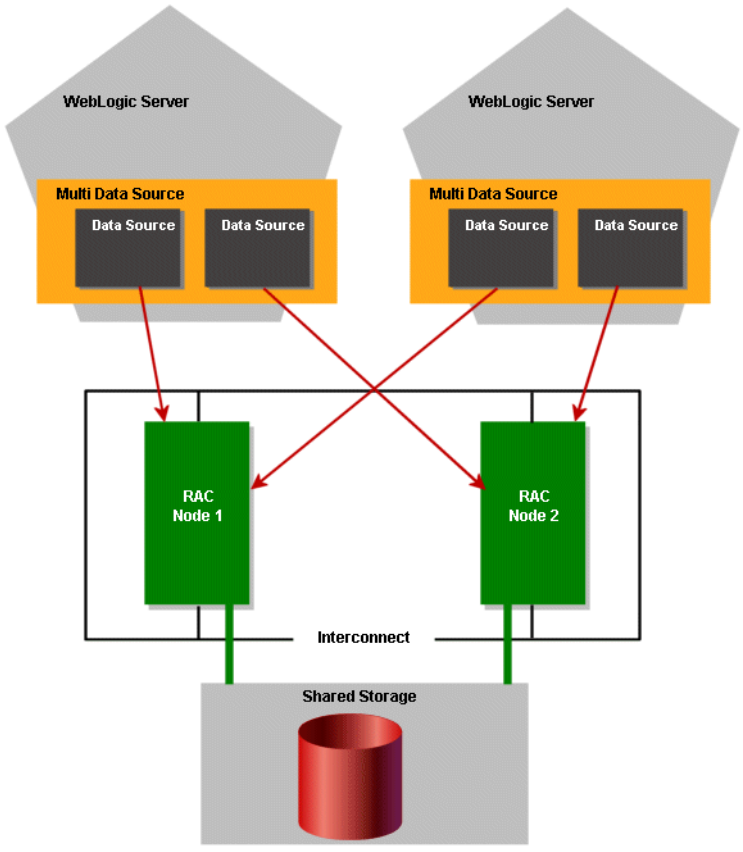
If a failure occurs during the `COMMIT` operation, the transaction manager attempts to retry the `COMMIT` operation several times. Note that the connection is blocked during these attempts. If the `COMMIT` operation is not successful during the first set of retry attempts, the application receives an exception. The transaction manager then continues to retry the `COMMIT` operation periodically until it is successful; if the transaction cannot be completed successfully within the abandon time period, the transaction is driven to completion heuristically.

Using Multi Data Sources with Oracle RAC

To connect WebLogic Server to multiple Oracle RAC nodes using multi data sources, first configure a JDBC data source for each RAC instance in your RAC cluster with the Oracle Thin driver. Then configure a multi data source, using either the algorithm for load balancing or the algorithm for failover, and add the data sources to it.

Figure B-3 shows a typical multi data source configuration.

Figure B-3 Multi Data Source Configuration



You can use the Administration Console or any other means that you prefer to configure your domain, such as the `weblogic.Admin` command line utility, the WebLogic Scripting Tool (WLST), or a JMX program. For information about configuring a WebLogic JDBC multi data source see Chapter 4, "Configuring JDBC Multi Data Sources."

To use a database connection in this configuration, your applications look up one multi data source on the JNDI tree and then request a connection. The multi data source determines which data source to use to satisfy the connection request based on the algorithm type specified in the configuration (that is, failover or load balancing).

Attributes of a Multi Data Source

The multi data source may have the following attributes, depending on the role of RAC in your system—load balancing or failover:

- `AlgorithmType="Load-Balancing"` or `AlgorithmType="Failover"`
  - With the Load-Balancing option, connection requests are distributed among available data sources; with the High-Availability option, connection requests are served by the first available pool in the list. When a data source becomes defunct, connection requests are served by the next data source in the list.
- `FailoverRequestIfBusy="true"`
  - With the Failover algorithm, this attribute enables failover when all connections in a data source are in use.
- `TestFrequencySeconds="120"`
  - This attribute controls the frequency at which WebLogic Server checks the health of data sources previously marked as unhealthy to see if connections can be recreated and if the data source can be re-enabled. For more details see Chapter 4, "Configuring JDBC Multi Data Sources."

Using Multi Data Sources with Global Transactions

In this configuration, a multi data source "pins" a transaction to one and only one Oracle RAC instance and failover is handled at the multi data source level when a RAC instance becomes unavailable. If there is a failure on a RAC instance before PREPARE, the operation is retried until the retry duration has expired. If there is a failure after PREPARE the transaction is failed over to another instance.

Rules for Data Sources within a Multi Data Source Using Global Transactions

The following rules apply to the XA data sources within a multi data source:

- All the data sources must be homogeneous. In other words, either all of them must use an XA driver or none of them can use an XA driver.
- If you choose to specify them, all XA-related attributes must be set to the same values for each data source. The attributes include the following:
  - `XARetryDurationSeconds`
  - `SupportsLocalTransaction`
  - `KeepXAConnTillTxComplete`
  - `NeedTxCtxOnClose`
  - `XAEndOnlyOnce`
  - `NewXAConnForCommit`
  - `RollbackLocalTxUponConnClose`
  - `RecoverOnlyOnce`
  - `KeepLogicalConnOpenOnRelease`

Note: If you are not using XA, BEA recommends the use of multi data sources for failover and load balancing across RAC instances, but the XA-specific configuration requirements above do not apply. For more information see "Using Multi Data Sources without Global Transactions" on page B-18.

Required Attributes of Data Sources within a Multi Data Source Using Global Transactions

Each data source within the multi data source should have the following attributes:

- Oracle JDBC Thin driver 10g. For example:

```
<url>jdbc:oracle:thin:@lcqsol24:1521:SNRAC1</url>
<driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
```

- `KeepXAConnTillTxComplete="true"`
  - Forces the data source to reserve a physical database connection and provide the same connection to an application throughout transaction processing until the distributed transaction is complete.
  - Required for proper transaction processing with Oracle RAC.
- `XARetryDurationSeconds="300"`

  - Enables the WebLogic Server transaction manager to retry XA recover, commit, and rollback calls for the specified amount of time.

- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the data source. See "Test Connections on Reserve to Enable Fail-Over" on page 4-4 for more details about this attribute.
  - Required to enable failover to another RAC node.
- `TestTableName="`*`name_of_small_table`*`"` The name of the table used to test a physical database connection. For more details about this attribute, see "Connection Testing Options for a Data Source" on page 3-23.

Sample Configuration Code

Sample configuration code for a multi data source and two associated data sources is shown below.

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>oracleRACXAPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@lcqsol24:1521:SNRAC1</url>
    <driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
    <profile-type>0</profile-type>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
```

```
    <jndi-name>oracleRACXAJndiName</jndi-name>
    <global-transactions-protocol>TwoPhaseCommit
        </global-transactions-protocol>
  </jdbc-data-source-params>
  <jdbc-xa-params>
    <keep-xa-conn-till-tx-complete>true</keep-xa-conn-till-tx-complete>
    <xa-end-only-once>true</xa-end-only-once>
    <xa-set-transaction-timeout>true</xa-set-transaction-timeout>
    <xa-transaction-timeout>120</xa-transaction-timeout>
    <xa-retry-duration-seconds>300</xa-retry-duration-seconds>
  </jdbc-xa-params>
</jdbc-data-source>
```

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>oracleRACXAPool2</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@lcqsol25:1521:SNRAC2</url>
    <driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
    <profile-type>0</profile-type>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>oracleRACXAJndiName2</jndi-name>
    <global-transactions-protocol>TwoPhaseCommit
        </global-transactions-protocol>
```

```
  </jdbc-data-source-params>
  <jdbc-xa-params>
    <keep-xa-conn-till-tx-complete>true</keep-xa-conn-till-tx-complete>
    <xa-end-only-once>true</xa-end-only-once>
    <xa-set-transaction-timeout>true</xa-set-transaction-timeout>
    <xa-transaction-timeout>120</xa-transaction-timeout>
    <xa-retry-duration-seconds>300</xa-retry-duration-seconds>
  </jdbc-xa-params>
</jdbc-data-source>
```

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>oracleRACXAMDS</name>
  <jdbc-data-source-params>
    <jndi-name>oracleRACMDSJndiName</jndi-name>
    <algorithm-type>Load-Balancing</algorithm-type>
    <data-source-list>oracleRACXAPool,oracleRACXAPool2</data-source-list>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Using Multi Data Sources without Global Transactions

The following sections describe a configuration that uses Oracle RAC with multi data sources in an application that does not require global transactions.

Attributes of Data Sources within a Multi Data Source Not Using Global Transactions

Data sources must have the following attributes:

- Oracle JDBC Thin driver 10g. For example:

  ```
  <url>jdbc:oracle:thin:@lcqsol24:1521:SNRAC1</url>
  <driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
  ```

- `TestConnectionsOnReserve="true"`

  - Enables testing of a database connection when an application reserves a connection from the data source. "Test Connections on Reserve to Enable Fail-Over" on page 4-4 for more details about this attribute.
  - Required to enable failover and connection request routing within a multi data source (effectively, failover to another RAC node).

- `TestTableName="name_of_small_table"`
  - The name of the table used to test a physical database connection. For more details about this attribute, see "Connection Testing Options for a Data Source" on page 3-23.

Sample Configuration Code

Sample configuration code for a WebLogic JDBC multi data source and associated data sources is shown below.

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>jdbcPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@lcqsol24:1521:snrac1</url>
    <driver-name>oracle.jdbc.OracleDriver</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-connections-on-reserve>true</test-connections-on-reserve>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>jdbcDataSource</jndi-name>
```

```
  </jdbc-data-source-params>
</jdbc-data-source>

<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>jdbcPool2</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@lcqsol25:1521:SNRAC2</url>
    <driver-name>oracle.jdbc.OracleDriver</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-connections-on-reserve>true</test-connections-on-reserve>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>jdbcDataSource2</jndi-name>
    <global-transactions-protocol>OnePhaseCommit
         </global-transactions-protocol>
  </jdbc-data-source-params>
</jdbc-data-source>

<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>jdbcNonXAMultiPool</name>
  <jdbc-data-source-params>
    <jndi-name>jdbcDataSource</jndi-name>
```

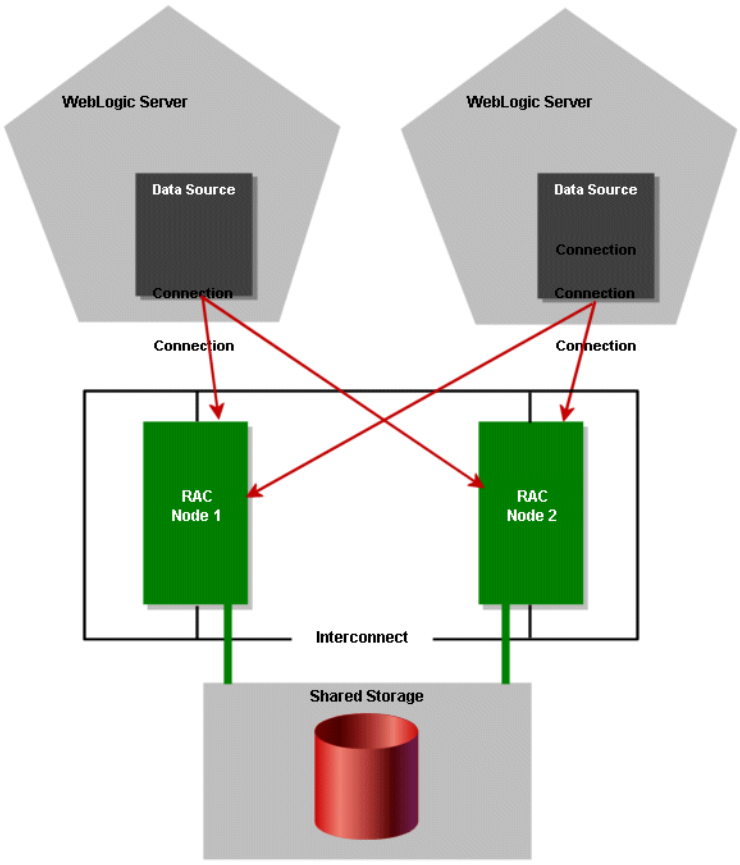
```
    <algorithm-type>Failover</algorithm-type>
    <data-source-list>jdbcPool,jdbcPool2</data-source-list>
    <failover-request-if-busy>true</failover-request-if-busy>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Note: Line breaks added for readability.

Using Connect-Time Failover with Oracle RAC

When multi data sources are not an option in your application, you can configure your data sources to use connect-time failover and load balancing. To connect WebLogic Server to multiple Oracle RAC nodes using data sources configured for connect-time failover and load balancing, configure a JDBC data source for each RAC instance in your RAC cluster with the Oracle Thin driver, as described in the sections that follow. Figure B-4 shows an overview of the system.

Figure B-4 Data Source Configuration with Oracle Thin Driver Connect-Time Failover



You can use the Administration Console or any other means that you prefer to configure your domain, such as the `weblogic.Admin` command line utility, the Weblogic Scripting Tool (WLST), or a JMX program.

When connections are created in the data source, the Oracle Thin driver determines which Oracle RAC instance to use. When an application gets a connection, it looks up a data source on the JNDI tree and requests a connection from the data source. The data source delivers one of the available connections from the pool of connections in the data source.

The following sections describe configuration options and requirements:

- Using Connect-Time Failover without Global Transactions

- XA Considerations and Limitations with Oracle RAC
- JDBC Store Recovery with Oracle RAC

Using Connect-Time Failover without Global Transactions

The following sections describe a configuration which uses Oracle RAC's connect-time failover features to handle connection failures. With this configuration, in some failure cases, the failover time is as long as the TCP timeout, which can be several minutes, depending on your environment.

Attributes of a Connect-Time Failover Configuration without Global Transactions

To use this configuration, create JDBC data sources in your WebLogic domain with the following attributes.

- Oracle JDBC Thin driver 10g configured for connect-time failover. For example:

```
<url>jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=
    (PROTOCOL=TCP)(HOST=lcqsol24)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)
    (HOST=lcqsol25)(PORT=1521))(FAILOVER=on)(LOAD_BALANCE=off))
    (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=snrac)))</url>
<driver-name>oracle.jdbc.OracleDriver</driver-name>
```

- `ConnectionReserveTimeoutSeconds="120"`
  - Enables application requests for a connection to wait 120 seconds for a connection to become available.
- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the data source. See "Test Connections on Reserve to Enable Fail-Over" on page 4-4 for more details about this attribute.
  - Required to enable failover to another RAC node.
- `TestTableName="name_of_small_table"` The name of the table used to test a physical database connection. For more details about this attribute, see "Connection Testing Options for a Data Source" on page 3-23.

Sample Configuration Code

Sample configuration code is shown below.

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/91"
  xmlns:sec="http://www.bea.com/ns/weblogic/91/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://www.bea.com/ns/weblogic/91/security/wls"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/91/domain.xsd">
  <name>oracleRACNonXAPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@(DESCRIPTION=
         (ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
         HOST=lcqsol24)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)
         (HOST=lcqsol25)(PORT=152))(FAILOVER=on)
         (LOAD_BALANCE=off))(CONNECT_DATA=(SERVER=DEDICATED)
         (SERVICE_NAME=snrac)))</url>
    <driver-name>oracle.jdbc.OracleDriver</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-connections-on-reserve>true</test-connections-on-reserve>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
    <profile-type>4</profile-type>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>oracleRACJndiName</jndi-name>
    <global-transactions-protocol>OnePhaseCommit
         </global-transactions-protocol>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Note: Line breaks added for readability.

Using Fast Connection Failover

WebLogic Server supports Fast Connection Failover, a Oracle feature which provides an application independent method to implement RAC event notifications, such a detection and cleanup of invalid connections, load balancing of available connections, and work redistribution on active RAC instances.

For more information, see Fast Connection Failover in the *Oracle® Database JDBC Developer's Guide and Reference*.

Required JDBC Driver Configuration for use with Oracle Fast Connection Failover

To enable Fast Connection Failover on an data source, set the following connection pool properties:

- In Driver Class Name—set the class name to `oracle.jdbc.pool.OracleDataSource`.
- In Properties—set the ONS configuration string to remotely subscribe the RAC nodes to Oracle FAN/ONS events. For example:
  `ods.ONSConfiguration("nodes=`*`hostname1`*`:`*`port1`*`,`*`hostname2`*`:`*`port2`*`")`

Note: Oracle's OracleDataSource class is not XA-capable, so the resulting data source does not implement a XA connection pool.

XA Considerations and Limitations with Oracle RAC

When using XA (global transactions) with Oracle RAC, consider the following requirements and limitations.

Required JDBC Driver Configuration for Use with XA

In this configuration, you must use the Oracle Thin driver connect-time failover to create database connections as described in "Using Connect-Time Failover without Global Transactions" on page B-23.

Oracle 9i RAC XA Requirements

Oracle 9i RAC has the following requirements when using XA.

A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the RAC Cluster

Global transactions must be initiated, prepared, and concluded in the same instance of the RAC cluster. WebLogic Server data sources manage this for you when you set `KeepXAConnTillTxComplete="true"` in the JDBC data source configuration.

Note: WebLogic Server relies on the connect-time failover feature in the Oracle Thin driver to work with Oracle RAC. As described in Oracle's Technical Note 235118.1, the Oracle Thin driver cannot guarantee that a transaction is initiated and concluded on the same RAC instance when the driver is configured for load balancing. As Oracle RAC requires that all database operations inside a global transaction be routed to the same Oracle instance, this known limitation means that you cannot use connect-time load balancing when using XA with Oracle RAC and, therefore, you cannot use a primary/primary RAC configuration.

Transaction IDs Must Be Unique Within the RAC Cluster

When using global transactions, transaction IDs (XIDs) must be unique within the RAC cluster. However, neither the Oracle Thin driver nor an Oracle RAC instance can determine if an XID is unique within the RAC cluster. Transactions with the same XID can execute SQL code on different instances of the RAC cluster without any exception.

The WebLogic Server Transaction Manager generates unique transaction IDs. However, in some failover scenarios, a transaction can continue on a RAC instance other than the originating instance, which can cause data inconsistencies. See “Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions” on page B-27.

Known Limitations When Using Oracle 9i RAC with WebLogic Server

The following sections describe known issues and limitations when using XA and WebLogic Server with Oracle RAC:

- “Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions” on page B-27
- “Potential for Data Deadlocks in Some Failure Scenarios” on page B-28
- “Potential for Transactions Completed Out of Sequence” on page B-28

Note: Some of these limitations are also described in Oracle's bug numbers 3428146 and 395790. Contact Oracle for more information about these issues.

Potential for Inconsistent Transaction Completion (Data Loss) in Some Failure Conditions

In some failure conditions, when multi data sources are not being used, transaction processing (data changes) that occurred on a RAC instance other than the instance on which a transaction was initiated ***will be lost without any notification or exception***.

For example, consider the following WebLogic Server configuration:

- A WebLogic cluster containing two servers: `server1` and `server2`.
- A JDBC data source `ds1` targeted to the cluster. (Identical instances of the data source are present on all instances in the cluster.)
- A JDBC data source `ds1` is configured to connect to an Oracle RAC cluster with connect-time failover enabled. `RAC1` is set as the primary RAC instance; `RAC2` is set as the secondary RAC instance. Data source `ds1` is targeted to the cluster. (Identical instances of the data source are present on all nodes in the WebLogic cluster.)

In the following scenario, some data changes will be lost:

1. Network connectivity between `server2` and `RAC1` is lost, which causes database connections in `cp1` on `server2` to fail over to `RAC2`. The same data source on `server1` still has connections to `RAC1`.
2. On `server1`, an application starts a transaction and uses a database connection from `cp1` (a connection to `RAC1`) to make data changes.
3. The application invokes an EJB on `server2`, which uses a database connection from `cp1` on `server2` (a connection to `RAC2`) to make data changes.
4. The application completes the transaction on `server1`.

Result: Data changes on RAC1 are committed. ***Data changes on RAC 2 are ignored***. The WebLogic Server transaction manager calls prepare and commit on the resource. In this case, because the data sources have the same name, they are considered to be the same resource, so the calls are made on only one instance of the data source. Because the data sources contain connections to different RAC instances, the data changes are committed on one RAC instance, but the changes on the other RAC instance are lost.

Workaround: Provide redundant network hardware between the WebLogic Server instance and the Oracle RAC instance to avoid the network failure.

Potential for Data Deadlocks in Some Failure Scenarios

There is a window of time in which transaction IDs are not available across the RAC cluster. Because of this known Oracle limitation, after some failure conditions, some incomplete transactions cannot be properly completed, which can result in deadlocks in the database. To prevent these failure conditions from arising, WebLogic Server provides two configuration attributes that enable XA call retry for Oracle RAC: `XARetryDurationSeconds` and `XARetryIntervalSeconds`. For more information about these configuration options, see "Delays During Failover" on page B-10.

Potential for Transactions Completed Out of Sequence

When using the Oracle DataBase Control, the order of transaction processing is not guaranteed. For example, if you implement a web service that uses DataBase Control do the following transaction sequence:

1. Create a table
2. Insert record 1
3. Insert record 2
4. Insert record 3
5. Select records

If the primary node goes down momentarily after the table is created, it is possible that transactions submitted to the database are performed out of sequence.

Known Issue Occurring After Database Server Crash

If, while a transaction is being processed, the database server instance crashes after the `PREPARE` operation is complete but before the results of that operation have been written to the transaction log, a `COMMIT` call from a client for that transaction may hang for several minutes and possibly until the TCP timeout period has expired. The window of time in which this might occur is small and the problem occurs rarely. There is no workaround for the issue at this time.

JDBC Store Recovery with Oracle RAC

If you are using a JDBC Store with Oracle RAC, there are features and limitations to consider that concern Oracle RAC node failover. See the following sections:

- Configuring a JDBC Store for Use with Oracle RAC
- Automatic Retry

Configuring a JDBC Store for Use with Oracle RAC

The way that a JDBC Store works limits the options you have for configuring one for use with Oracle RAC. You cannot configure a JDBC store to use a JDBC data source that is configured to support global transactions. The JDBC store must use a JDBC data source that uses a non-XA JDBC driver.

A JDBC Store holds on to a connection until that connection fails, at which point it moves on to the next connection and repeats the process. Therefore you cannot implement load balancing with a JDBC Store, including using a load balancing multi data source. You should configure a multi data source for a JDBC store to use the Failover algorithm. For more information about this configuration option, see "Using Multi Data Sources without Global Transactions" on page B-18.

Automatic Retry

JMS has a limited connection retry mechanism which enables it to silently react to the failure of the RAC node that hosts its database connection. If the database has experienced either a minor network 'hiccup' or a RAC database has failed over to another node, the second connection attempt (the retry) will succeed to the next RAC node.

The time within which this retry is attempted and the number of retries attempted are limited to minimize the negative effects that an extended connection retry time could cause. If the database connection remains unavailable for a long period of time, the delay can impede the ability of JMS to properly continue its processing (for example, to maintain proper message ordering). Also, the transaction manager could declare the JMS resource of a transaction to be dead if there is not enough processing progress made within this time period, or out-of memory conditions could arise. There are system-level tuning guidelines that can help minimize the RAC failover time frame which is critical to the success of the automatic retry.

The tight loop on the automatic retry is particularly important when JMS processing occurs with transactions. If an I/O failure occurs in the JDBC Store, the store record is in an unknown state which will put the message itself in an unknown state. To prevent the message from being committed in this unknown state, JMS will mark the transaction associated with the message as a "failedTransaction." Any future attempts by the transaction manager to finishing committing the message will cause JMS to throw a javax.transaction.xa.XAException with an errorCode set to `XAException.XAER_RMERR`. This exception is an indication to the transaction manager that a transient error has occurred in the resource manager (JMS) and that the transaction manager

should retry commit processing. The retry logic provides a second attempt to establish the connection before JMS communicates any failure to the upper layer which would translate into an RMERR. If the RMERR is generated, then the only way to recover the message and complete the transaction is to restart WebLogic Server.

The automatic retry logic is currently governed by an option on WebLogic Server as follows:

```
-Dweblogic.store.jdbc.IORetryDelayMillis=X
```

Where `X` is the number of milliseconds that should elapse before the connection to the database is retried. The default value is 1000 milliseconds. This value is restricted to the range `0` to `15000` milliseconds, and the retry will only be attempted once. If a failure occurs on the second attempt, an exception will be propagated up the call stack and a manual restart will be required to recover the messages associated with the failed transaction.

Note: In the event that an automatic retry attempt is not successful, you must restart WebLogic Server.