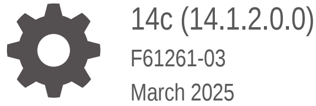
Oracle® Fusion Middleware

Administering JDBC Data Sources for Oracle WebLogic Server



14c (14.1.2.0.0)
F61261-03
March 2025

Oracle Fusion Middleware Administering JDBC Data Sources for Oracle WebLogic Server, 14c (14.1.2.0.0)

F61261-03

Copyright © 2007, 2025, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

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

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

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

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

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

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

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

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

Contents

Preface

Audience xv
Documentation Accessibility xv
Diversity and Inclusion xv
Related Documentation xvi
Conventions xvi

1 About WebLogic JDBC Resources

JDBC Resources 1-1
JDBC Data Sources 1-2
JMX and WLST Access for JDBC Resources 1-3
WebLogic Server with Oracle RAC 1-3
Advanced Configurations for Oracle Drivers and Databases 1-3

2 Configuring WebLogic JDBC Resources

JDBC System Modules 2-1
    Generic Data Source Modules 2-2
    Active GridLink Data Source System Modules 2-3
    Multi Data Source System Modules 2-3
JDBC Application Modules 2-4
    Standard Jakarta EE Application Modules 2-4
    Proprietary JDBC Application Modules 2-4
        Including Drivers in EAR/WAR Files 2-5
JDBC Module File Naming Requirements 2-6
JDBC Modules in Versioned Applications 2-6
JDBC Schema 2-7
JDBC Data Source Type 2-7
JMX and WLST Access for JDBC Resources 2-7
    JDBC MBeans for System Resources 2-8
    JDBC Management Objects in the Jakarta Management Model (the Jakarta Management Specification Support) 2-9
    Using WLST to Create JDBC System Resources 2-9

How to Modify and Monitor JDBC Resources 2-11
Best Practices when Using WLST to Configure JDBC Resources 2-11
Creating High-Availability JDBC Resources 2-11

3 Configure Database Connectivity

Using JDBC Drivers with WebLogic Server 3-1
Types of JDBC Drivers 3-1
JDBC Driver Support 3-5
JDBC Drivers Installed with WebLogic Server 3-5
Adding Third-Party JDBC Drivers Not Installed with WebLogic Server 3-6
Globalization Support for the Oracle Thin Driver 3-9
Using Oracle JDBC Driver Extensions 3-9
Using OCI Vault 3-10
Storing Wallet and DB Connection String 3-11
JDBC Diagnosability 3-13
Enabling JDBC Driver Logging 3-13
Diagnosing First Failure 3-14
Configuring JDBC Data Sources 3-15
Creating a JDBC Data Source 3-15
Configure JDBC Data Source Properties 3-16
Configure Transaction Options 3-17
Configure Connection Properties 3-17
Configure Testing Options 3-17
Target JDBC Data Sources 3-18
Configuring Connection Pool Features 3-18
Enabling JDBC Driver-Level Features 3-18
Enabling Connection-based System Properties 3-19
Enabling Connection-based Encrypted Properties 3-20
Initializing Database Connections with SQL Code 3-20
Advanced Connection Properties 3-21
Define Fatal Error Codes 3-21
Using Edition-Based Redefinition 3-21
Configure Oracle Parameters 3-24
Configure ONS Client Parameters 3-24
Tuning Generic Data Source Connection Pools 3-24
Generic Data Source Handling for Oracle RAC Outages 3-24
Generic Data Source Handling of Driver-Level Failover 3-25

4 JDBC Data Sources Types

Using the Default Data Source 4-1

What is Default Data Source 4-1
Defining a Custom Default Data Source 4-3
Compatibility Limitations When Using a Default Data Source 4-4
Using Generic Data Sources 4-4
What is Generic Data Source 4-4
Configuring Generic Data Source 4-4
Using JDBC Multi Data Sources 4-5
What is Multi Data Source 4-6
Adding a Database Node 4-7
Removing a Database Node 4-7
Configuring Multi Data Sources 4-7
Choosing the Multi Data Source Algorithm 4-8
Multi Data Source Fail-Over Limitations and Requirements 4-9
Controlling Multi Data Source Failover with a Callback 4-10
Deploying JDBC Multi Data Sources on Servers and Clusters 4-12
Multi Data Source Failover Enhancements 4-12
Connection Request Routing Enhancements When a Generic Data Source Fails 4-12
Automatic Re-enablement on Recovery of a Failed Generic Data Source within a Multi Data Source 4-13
Enabling Failover for Busy Generic Data Sources in a Multi Data Source 4-13
Controlling Multi Data Source Failback with a Callback 4-13
Planned Database Maintenance with a Multi Data Source 4-15
Shutting Down the Data Source 4-15
Using Active GridLink Data Sources 4-18
What is Active GridLink Data Source 4-18
Fast Connection Failover 4-20
Runtime Connection Load Balancing 4-21
GridLink Affinity 4-22
SCAN Addresses 4-24
Secure Communication using Oracle Wallet with ONS Listener 4-25
Support for Active Data Guard 4-25
Supported Oracle On-Premises and Cloud Database Services 4-25
Using Socket Direct Protocol 4-26
Configuring Active GridLink Data Source 4-26
Configure JDBC Data Source Properties 4-27
Configure Transaction Options 4-27
Configure Connection Properties 4-27
Test Connections 4-29
Configure ONS Client 4-29
Target the Data Source 4-31
Configuring Oracle Parameters 4-31
Configuring an ONS Client Using WLST 4-31

Configuring Runtime Load Balancing using SDP 4-32
Configuring Active GridLink Connection Pool Features 4-32
Enabling JDBC Driver-Level Features 4-33
Enabling Connection-based System Properties 4-33
Initializing Database Connections with SQL Code 4-33
Tuning Active GridLink Data Source Connection Pools 4-34
Monitoring Active GridLink JDBC Resources 4-34
Viewing Run-Time Statistics 4-34
Debug Active GridLink Data Sources 4-35
Using Active GridLink Data Sources without FAN Notification 4-36
Best Practices for Active GridLink Data Sources 4-37
Catch and Handle Exceptions 4-37
Connection Creation with Active GridLink Data Sources 4-38
Comparing Active GridLink and Multi Data Sources 4-38
Migrating from Multi Data Source to Active GridLink 4-39
Application Changes to Migrate a Multi Data Source 4-39
Configuration Changes to Migrate a Multi Data Source 4-39
Basic Migration Steps 4-40
Managing Database Downtime with Active GridLink Data Sources 4-40
Active GridLink Configuration for Database Outages 4-40
Planned Outage Procedures 4-41
Unplanned Outages 4-45
Gradual Draining 4-45
Using Universal Connection Pool Data Sources 4-47
What is Universal Connection Pool Data Source 4-48
Creating a Universal Connection Pool Data Source 4-49
Configuring a UCP Data Source in the WebLogic Remote Console 4-49
Configuring a UCP Using WLST 4-54
Universal Connection Pool Multi Tenant Shared Pool support 4-55
Monitoring Universal Connection Pool JDBC Resources 4-57
Oracle Sharding Support 4-57

5 JDBC Data Source Transaction Options

Enabling Support for Global Transactions with a Non-XA JDBC Driver 5-2
Understanding the Logging Last Resource Transaction Option 5-2
Advantages to Using the Logging Last Resource Optimization 5-3
Enabling the Logging Last Resource Transaction Optimization 5-4
Programming Considerations and Limitations for LLR Data Sources 5-4
Administrative Considerations and Limitations for LLR Data Sources 5-5
Understanding the Emulate Two-Phase Commit Transaction Option 5-6
Limitations and Risks When Emulating Two-Phase Commit Using a Non-XA Driver 5-7

Heuristic Completions and Data Inconsistency 5-7
Cannot Recover Pending Transactions 5-8
Possible Performance Loss with Non-XA Resources in Multi-Server Configurations 5-8
Multiple Non-XA Participants 5-8
Local Transaction Completion when Closing a Connection 5-8

6 Advanced Configurations for Oracle Drivers and Databases

JDBC Replay Driver 6-1
How JDBC Replay Driver Works 6-2
Requirements and Considerations 6-4
Configuring JDBC Replay Driver 6-4
Selecting the Driver for JDBC Replay Driver 6-5
Using a Connection Callback 6-5
Setting the Replay Timeout 6-6
Disabling JDBC Replay Driver for a Connection 6-6
Configuring Logging for JDBC Replay Driver 6-7
Viewing Runtime Statistics for JDBC Replay Driver 6-7
JDBC Replay Driver Auditing 6-10
Limitations with JDBC Replay Driver with Oracle 12c Database 6-11
Database Resident Connection Pooling 6-11
Requirements and Considerations 6-11
Configuring DRCP 6-12
Configuring a Data Source for DRCP 6-12
Configuring a Database for DRCP 6-13
Global Data Services 6-13
Requirements and Considerations 6-14
Creating a Active GridLink Data Source for GDS Connectivity 6-14
Container Database with Pluggable Databases 6-14
Creating Service for PDB Access 6-15
DRCP and CDB/PDB 6-15
Setting the PDB using JDBC 6-15
Service Switching 6-16

7 Using Connection Harvesting

What is Connection Harvesting 7-1
Enable Connection Harvesting 7-2
Marking Connections Harvestable 7-2
Recover Harvested Connections 7-2

8 Using Shared Pooling Data Sources

How Shared Pooling Works 8-1
Requirements and Considerations when using Shared Pooling Data Sources 8-1
Configuring Shared Pooling 8-2
    Configuring WebLogic Server-Specific Driver Properties for Shared Pooling 8-2
    Configuring Database for Shared Pooling 8-4

9 Using Oracle Databases with WebLogic Server

WebLogic JDBC Features for Oracle Database 23ai 9-1
WebLogic JDBC Features for Oracle Database 19.3 9-1
    Support for JDBC 4.3 Interfaces 9-2
WebLogic JDBC Features for Oracle Database 12.2 9-2
    JDBC 4.2 Interfaces 9-3
    Database 12.2 JDBC Replay Driver 9-4
    AGL Support for URL with @alias or @ldap 9-4
WebLogic JDBC Features for Oracle Database 12.1 9-4
    JDBC 4.1 Support for JDK 7 9-6
    JDBC Replay Driver Support 9-6
    Database Resident Connection Pooling Support 9-6
    Container Database with Pluggable Databases 9-6
    Global Data Services Support 9-6
    Automatic ONS Listeners 9-6

10 Labeling Connections

What is Connection Labeling 10-1
Implementing Labeling Callbacks 10-2
Creating a Labeling Callback 10-2
    Example Labeling Callback 10-3
Registering a Labeling Callback 10-4
    Removing a Labeling Callback 10-5
    Applying Connection Labels 10-5
Reserving Labeled Connections 10-6
Checking Unmatched labels 10-7
Removing a Connection Label 10-7
Using Initialization and Reinitialization Costs to Select Connections 10-7
    Considerations When Using Initialization and Reinitialization Costs 10-8
Using Connection Labeling with Packaged Applications 10-8

11 Understanding Data Source Security

About WebLogic Data Source Security Options 11-1
WebLogic Data Source Security Options 11-2
- Credential Mapping vs. Database Credentials 11-3
- Set Client Identifier on Connection 11-5
- Oracle Proxy Session 11-6
- Identity-based Connection Pooling 11-8

Connections within Transactions 11-9
WebLogic Data Source Resource Permissions 11-9
Data Source Security Example 11-10
Using Encrypted Connection Properties 11-12
- Best Practices 11-12
- WLST Examples 11-13

Using SSL and Encryption with Data Sources and Oracle Drivers 11-13
- Using SSL with Data Sources and Oracle Drivers 11-14
  - Using SSL with Oracle Wallet 11-14
  - Active GridLink ONS over SSL 11-15
- Using Data Encryption with Data Sources and Oracle Drivers 11-15

12 Creating and Managing Oracle Wallet

What Is Oracle Wallet 12-1
Where to Keep Your Wallet 12-1
How to Create an External Password Store 12-2
Define a WebLogic Server Data Source Using a Wallet 12-3
- Copy the Wallet Files 12-3
- Update the Data Source Configuration 12-3

Use a TNS Alias Instead of a DB Connection String 12-3
Use DBClientData Modules for Portability 12-4
- What Are DBClientData Modules 12-4
- Why Use DBClientData Modules 12-5
- Manage DBClientData Modules 12-5
  - Deploy DBClientData Modules 12-5
  - Distribute DBClientData Modules 12-6
  - Redeploy DBClientData Modules 12-7
- Configure a Data Source to Use DBClientData Modules 12-7

13 Deploying Data Sources on Servers and Clusters

Deploying Data Sources on Servers and Clusters 13-1

Minimizing Server Startup Hang Caused By an Unresponsive Database 13-1

14 Using WebLogic Server with Oracle RAC

Overview of Oracle Real Application Clusters 14-1
Software Requirements 14-2
JDBC Driver Requirements 14-2
Hardware Requirements 14-2
Configuration Options in WebLogic Server with Oracle RAC 14-3
Choosing a WebLogic Server Configuration for Use with Oracle RAC 14-3
Validating Connections when using WebLogic Server with Oracle RAC 14-4
Additional Considerations When Using WebLogic Server with Oracle RAC 14-4

15 Using JDBC Drivers with WebLogic Server

JDBC Driver Support 15-1
JDBC Drivers Installed with WebLogic Server 15-1
Adding Third-Party JDBC Drivers Not Installed with WebLogic Server 15-2
Globalization Support for the Oracle Thin Driver 15-6

16 Monitoring WebLogic JDBC Resources

Viewing Run-Time Statistics 16-1
Data Source Statistics 16-1
Prepared Statement Cache Statistics 16-1
Profile Logging 16-2
Collecting Profile Information 16-2
Profile Types 16-2
Connection Usage (WEBLOGIC.JDBC.CONN.USAGE) 16-3
Connection Reservation Wait (WEBLOGIC.JDBC.CONN.RESV.WAIT) 16-3
Connection Reservation Failed (WEBLOGIC.JDBC.CONN.RESV.FAIL) 16-3
Connection Leak (WEBLOGIC.JDBC.CONN.LEAK) 16-4
Connection Last Usage (WEBLOGIC.JDBC.CONN.LAST_USAGE) 16-4
Connection Multithreaded Usage (WEBLOGIC.JDBC.CONN.MT_USAGE) 16-4
Statement Cache Entry (WEBLOGIC.JDBC.STMT_CACHE.ENTRY) 16-5
Statements Usage (WEBLOGIC.JDBC.STMT.USAGE) 16-5
Connection Unwrap (WEBLOGIC.JDBC.CONN.UNWRAP) 16-5
JDBC Object Closed Usage (WEBLOGIC.JDBC.CLOSED_USAGE) 16-5
Local Transaction Connection Leak (WEBLOGIC.JDBC.CONN.LOCALTX_LEAK) 16-6
Example Profile Information Record Log 16-6
Accessing Diagnostic Data 16-6
Callbacks for Monitoring Driver-Level Statistics 16-7

Debugging JDBC Data Sources 16-7
Enabling Debugging 16-7
Enable Debugging Using the Command Line 16-7
Enable Debugging Using the WebLogic Remote Console 16-8
Enable Debugging Using the WebLogic Scripting Tool 16-8
Changes to the config.xml File 16-9
JDBC Debugging Scopes 16-9
Set Debugging for UCP or ONS 16-10
Request Dyeing 16-11

17 Managing WebLogic JDBC Resources

Testing Data Sources and Database Connections 17-1
Managing the Statement Cache for a Data Source 17-2
Clearing the Statement Cache for a Data Source 17-2
Clearing the Statement Cache for a Single Connection 17-2
Shrinking a Connection Pool 17-3
Resetting a Connection Pool 17-3
Suspending a Connection Pool 17-4
Resuming a Connection Pool 17-5
Shutting Down a Data Source 17-5
Starting a Data Source 17-6
Managing DBMS Network Failures 17-6

18 Tuning Data Source Connection Pools

Increasing Performance with the Statement Cache 18-2
Statement Cache Algorithms 18-2
LRU (Least Recently Used) 18-2
Fixed 18-3
Statement Cache Size 18-3
Usage Restrictions for the Statement Cache 18-3
Calling a Stored Statement After a Database Change May Cause Errors 18-4
Using setNull In a Prepared Statement 18-4
Statements in the Cache May Reserve Database Cursors 18-4
Other Considerations When Using the Statement Cache 18-5
Initial Capacity Enhancement in the Connection Pool 18-5
Connection Testing Options for a Data Source 18-6
Database Connection Testing Semantics 18-7
Connection Testing When Database Connections are Created 18-8
Periodic Connection Testing 18-8
Testing Reserved Connections 18-8

Minimizing Connection Test Delay After Database Connectivity Loss 18-9
Minimizing Connection Request Delays After Loss of DBMS Connectivity 18-10
Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection 18-10
Database Connection Testing Configuration Recommendations 18-11
Database Connection Testing Using Default Test Table Name 18-12
Database Connection Testing Options 18-12
Enabling Connection Creation Retries 18-12
Enabling Connection Requests to Wait for a Connection 18-13
Connection Reserve Timeout 18-13
Limiting the Number of Waiting Connection Requests 18-13
Automatically Recovering Leaked Connections 18-13
Avoiding Server Lockup with the Correct Number of Connections 18-14
Limiting Statement Processing Time with Statement Timeout 18-14
Using Pinned-To-Thread Property to Increase Performance 18-14
Changes to Connection Pool Administration Operations When PinnedToThread is Enabled 18-15
Additional Database Resource Costs When PinnedToThread is Enabled 18-16
Using Unwrapped Data Type Objects 18-16
How to Disable Wrapping 18-17
Disable Wrapping using the Remote Console 18-17
Disable Wrapping using WLST 18-18
Tuning Maintenance Timers 18-18
JDBC Connection Creation Limits 18-18

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 Remote Console A-2
JDBC Packaged Module Requirements A-3
JDBC Application Module Limitations A-3
Creating a Generic Data Source Module A-3
Creating an Active GridLink Data Source Module A-5
Creating a Multi Data Source Module A-5
Encrypting Database Passwords in a JDBC Module A-5
Deploying JDBC Modules to New Domains A-5
Application Scoping for a Packaged JDBC Module A-6
Referencing a JDBC Module in Jakarta EE Descriptor Files A-6
Packaged JDBC Module References in weblogic-application.xml A-7
Packaged JDBC Module References in Other Descriptors A-8
Packaging an Enterprise Application with a JDBC Module A-8
Deploying an Enterprise Application with a JDBC Module A-9

Getting a Database Connection from a Packaged JDBC Module A-9

B Using Multi Data Sources with Oracle RAC

Overview of Oracle RAC B-2
Oracle RAC Scalability with WebLogic Server Multi Data Sources B-3
Oracle RAC Availability with WebLogic Server Multi Data Sources B-3
Oracle RAC Load Balancing with WebLogic Server Multi Data Sources B-3
Software Requirements B-3
JDBC Driver Requirements B-4
Hardware Requirements B-4
WebLogic Server Cluster B-4
Oracle RAC Cluster B-4
Shared Storage B-4
Configuring Multi Data Sources with Oracle RAC B-5
Choosing a Multi Data Source Configuration for Use with Oracle RAC B-5
Configuring Multi Data Sources for use with Oracle RAC B-5
Attributes of a Multi Data Source B-6
Configuration Considerations for Failover B-7
Multi Data Source-Managed Failover and Load Balancing B-7
Delays During Failover B-7
Failure Handling Walkthrough for Global Transactions B-8
Configuring the Listener Process for Each Oracle RAC Instance B-9
Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled B-10
Additional Configuration Considerations B-11
Using Multi Data Sources with Global Transactions B-12
Rules for Data Sources within a Multi Data Source Using Global Transactions B-12
Required Attributes of Data Sources within a Multi Data Source Using Global Transactions B-13
Sample Configuration Code B-13
Using Multi Data Sources without Global Transactions B-15
Attributes of Data Sources within a Multi Data Source Not Using Global Transactions B-15
Sample Configuration Code B-15
Configuring Connections to Services on Oracle RAC Nodes B-17
Configuring a Data Source to Connect to a Service B-17
Service Connection Configurations B-18
Workload Management B-18
Load Balancing B-20
Connection Pool Capacity Planning B-22
Using SCAN Addresses with Multi Data Sources B-25
XA Considerations and Limitations when using Multi Data Sources with Oracle RAC B-25
Oracle RAC XA Requirements when using Multi Data Sources B-26

Known Issue Occurring After Database Server Crash B-26
JDBC Store Recovery with Oracle RAC B-26
Configuring a JDBC Store for Use with Oracle RAC B-26
Automatic Retry for JMS Connections B-27

Preface

This document contains Java Database Connectivity (JDBC) data source configuration and administration information.

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Documentation
- Conventions

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.

It is assumed that the reader is familiar with Jakarta EE and JDBC concepts. This document emphasizes the value-added features provided by WebLogic Server.

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve.

Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

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

- *Developing JDBC Applications for Oracle WebLogic Server* is a guide to JDBC API programming with WebLogic Server.
- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications in development and production environments.

JDBC Samples and Tutorials

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

JDBC Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in `EXAMPLES_HOME\wl_server\examples\src\examples`, where `EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

For more information, see Sample Applications and Code Examples

New WebLogic Server Features

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

Conventions

The following text conventions are used in this document:

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

1 About WebLogic JDBC Resources

To configure JDBC resources you need to understand how to use the different types of data sources available such as Active GridLink (AGL) and Multi Data Source (MDS). 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.

- JDBC Resources
- JDBC Data Sources
- JMX and WLST Access for JDBC Resources
- WebLogic Server with Oracle RAC
- Advanced Configurations for Oracle Drivers and Databases

JDBC Resources

The key to understanding WebLogic JDBC data source configuration is to understand *who* creates a JDBC resource or *how* a JDBC resource is created and managed. This determines how a resource will be deployed and modified.
Both system administrators and programmers can create and manage JDBC resources either as system modules or as application modules. WebLogic supports either standard or proprietary JDBC application modules. Regardless of whether you are using JDBC system modules or JDBC application modules, each JDBC data source is represented by an XML file (a module).

- **System Modules:** WebLogic Administrators typically use the WebLogic Remote 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.
- **Application Modules:** Programmers create modules in a development tool that supports creating an XML descriptor file, then package the JDBC modules with an application (for example, an EAR or WAR file) and pass the application to a WebLogic Administrator to deploy. These JDBC modules are considered *application modules*. See JDBC Application Modules.

The standard JDBC application modules are created using the JEE 6 annotations or schema definitions based on `datasourcedefinition`. The proprietary JDBC application modules are a WebLogic-specific extension of Jakarta EE modules and can be configured either within a Jakarta EE application or as stand-alone modules.

These documents conform to the `jdbc-data-source.xsd` schema (available at `http://www.oracle.com/webfolder/technetwork/weblogic/jdbc-data-source/index.html`).

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

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

Module Type	Created with	Add/ Remove Modules with Remote Console	Modify with JMX (remotely)	Modify with JSR-88 (non-remotely)	Modify with Remote Console
System	WebLogic Remote Console or WLST	Yes	Yes	No	Yes—via JMX
Application	Oracle Enterprise Pack for Eclipse (OEPE), Oracle JDeveloper, another IDE, or an XML editor	No	No	Yes—via a deployment plan	Yes—via a deployment plan

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.

You can configure database connectivity by adding JDBC data sources to your WebLogic domain. Configuring data sources requires several steps including choosing a type of data source, creating the data source, configuring connection pools and Oracle database parameters and so on. See Configuring JDBC Data Sources.

Types of WebLogic Server JDBC Data Sources

WebLogic Server provides the following types of data sources:

- **Default data sources**—Oracle provides a default data source required by a Jakarta EE 8-compliant runtime. Applications can use this pre-configured data source to access the Derby database installed with the WebLogic Server. See Using the Default Data Source.
- **Generic data sources**—Generic data sources and their connection pools provide connection management processes that help keep your system running efficiently. You can set options in the data source to suit your applications and your environment. See Using Generic Data Sources.
- **Active GridLink data sources**—A data source that provides a connection pool that spans one or more nodes in one or more Oracle RAC clusters. It supports dynamic load balancing of connections across the nodes and handles events indicating nodes added and removed from the cluster(s). See Using Active GridLink Data Sources.
- **Multi Data Source** —A MDS is an abstraction around a group of Generic data sources that provides load balancing or failover processing. See Configuring JDBC Multi Data Sources.
- **Universal Connection Pool (UCP) data source**—A UCP data source provides an option for users who wish to use Oracle Universal Connection Pooling to connect to Oracle

Databases. UCP provides an alternative connection pooling technology to Oracle WebLogic Server connection pooling. See Using Universal Connection Pool Data Sources.

JMX and WLST Access for JDBC Resources

You can create JDBC resources using any of the WebLogic Server administration tools. When you create JDBC resources, WebLogic Server creates MBeans (Managed Beans) for each of the resources. You can then access these MBeans using Java Management Extensions (JMX) or the WebLogic Scripting Tool (WLST).
The WebLogic Scripting Tool is a complete, command-line scripting environment for managing Oracle WebLogic Server domains, based on the Java scripting interpreter, Jython. In addition to supporting standard Jython features such as local variables, conditional variables, and flow control statements, the WebLogic Scripting Tool provides a set of scripting functions (commands) that are specific to Oracle WebLogic Server. You can extend the WebLogic scripting language to suit your needs by following the Jython language syntax. See, Using the WebLogic Scripting Tool in *Understanding the WebLogic Scripting Tool*.

To integrate third-party management systems with the WebLogic Server management system, WebLogic Server provides standards-based interfaces that are fully compliant with the JMX specification. Software vendors can use these interfaces to monitor WebLogic Server MBeans, to change the configuration of a WebLogic Server domain, and to monitor the distribution (activation) of those changes to all server instances in the domain. See, Understanding WebLogic Server MBeans in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

For a complete list of WebLogic Server administration tools, see Summary of System Administration Tools and APIs in *Understanding Oracle WebLogic Server*.

For detailed information, see JMX and WLST Access for JDBC Resources.

WebLogic Server with Oracle RAC

Oracle WebLogic Server provides strong support for Oracle Real Application Clusters (RAC), minimizing database access time while allowing transparent access to rich pooling management functions that maximizes both connection performance and availability.
See:

- Using WebLogic Server with Oracle RAC
- Using Multi Data Sources with Oracle RAC
- Using Fast Connection Failover with Oracle RAC

Advanced Configurations for Oracle Drivers and Databases

Oracle provides advanced configuration options such as JDBC Replay Driver, database resident connection policy, and global database services to improve data source and driver performance when using Oracle drivers and databases. These configuration options help in the management of connection reservations in the data source.
For more information, see Advanced Configurations for Oracle Drivers and Databases.

2 Configuring WebLogic JDBC Resources

To configure the JDBC resource you need to understand how to use JDBC resources in a WebLogic domain, ownership of resources, how to create MBeans for JDBC resources using tools like JMX and WLST, and how to increase the availability of JDBC resources. In WebLogic Server, you can configure database connectivity by configuring JDBC resources and then targeting or deploying the JDBC resources to servers or clusters in your WebLogic domain.

- JDBC System Modules
  When you create a JDBC resource (data source) using the WebLogic Remote 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.
- JDBC Application Modules
  In contrast to system resource modules, the developer creates, packs, and owns the JDBC modules whereas the Administrator only deploys the module.
- JDBC Module File Naming Requirements
  All WebLogic JDBC module files must end with the `-jdbc.xml` suffix, such as `examples-demo-jdbc.xml`.
- JDBC Modules in Versioned Applications
  WebLogic Server identifies the data source defined in the JDBC module with a specific name.
- JDBC Schema
  In support of the modular deployment model for JDBC resources in WebLogic Server, Oracle 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.
- JDBC Data Source Type
  Data sources should have a `datasource-type` set in the descriptor. This functionality was added in WebLogic Server 12.2.1 and is optional for backward compatibility.
- JMX and WLST Access for JDBC Resources
  This section describes how to access WebLogic Server MBeans using JMX client or the WebLogic Scripting Tool (WLST).
- Creating High-Availability JDBC Resources
  To improve the availability of your JDBC resource and load balance communication between resources you can target or deploy a JDBC data source to the members of a cluster using the WebLogic Remote Console.

JDBC System Modules

When you create a JDBC resource (data source) using the WebLogic Remote 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 `jdbc-data-source.xsd` schema (available at `http://www.oracle.com/webfolder/technetwork/weblogic/jdbc-data-source/index.html`).

JDBC data sources that you configure this way are known as *system modules*. Administrator owns the system modules and can delete, modify, or add similar resources at any time. System modules are globally available for targeting 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`.

- Generic Data Source Modules
- Active GridLink Data Source System Modules
- Multi Data Source System Modules

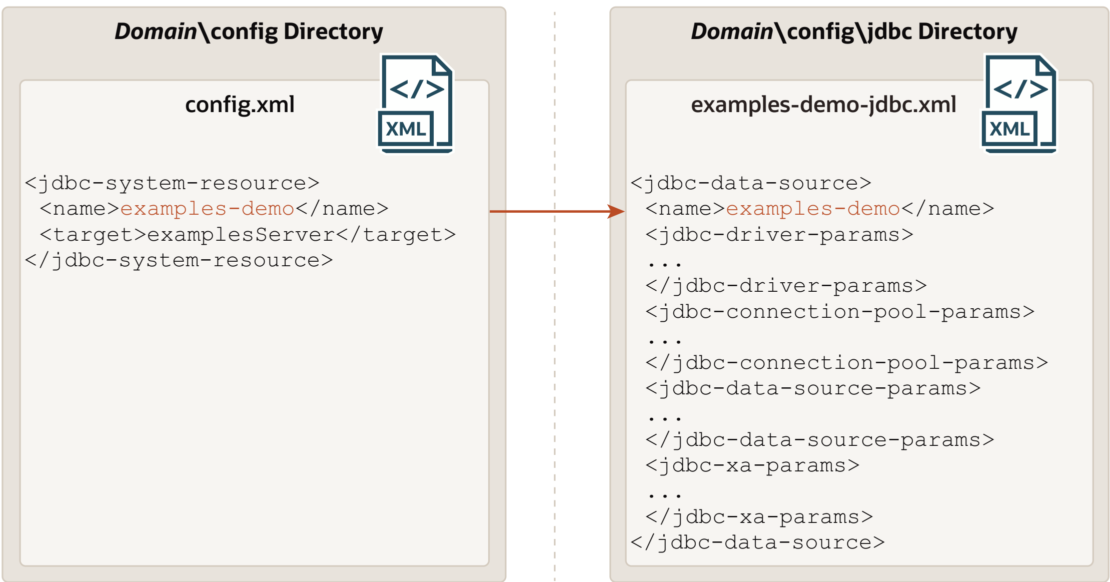
Generic Data Source Modules

Generic 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 2-1 shows an example of a data source listing in a `config.xml` file and the module that it maps to.

Note:

Generic is the term used to distinguish a simple data source from a Multi Data Source or Active GridLink data source.

Figure 2-1 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-jdbc.xml` module in the *domain*`\config\jdbc` folder.

Active GridLink Data Source System Modules

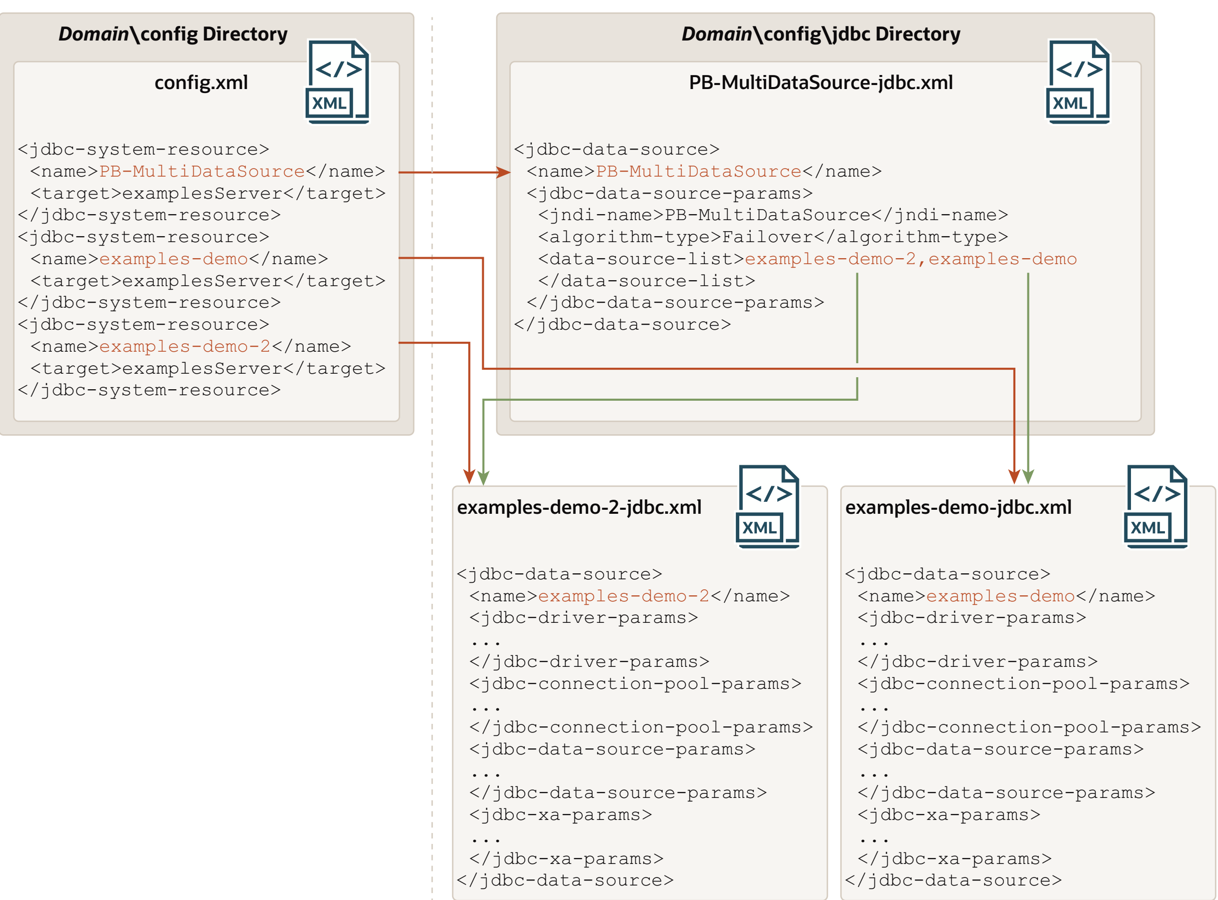
Active GridLink data source system modules are included in the domain's `config.xml` file as a `jdbc-system-resource` element, similar to Generic data source system modules. Active GridLink data sources include a `jdbc-oracle-params` section that includes `ONS` and `FAN`.

For more information about Active GridLink data sources, see Using Active GridLink Data Sources.

Multi Data Source System Modules

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

Figure 2-2 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 MDS and the two Generic data sources used by the MDS, which are also listed within the MDS module. Your application can look up any of these modules on the JNDI tree and request a database connection. If you look up the MDS, the MDS determines which of the Generic 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.

> **Note:**
>
> Members of a MDS must be Generic data sources; they cannot be MDS or Active GridLink data sources.

For more information about MDS, see Configuring JDBC Multi Data Sources.

JDBC Application Modules

In contrast to system resource modules, the developer creates, packs, and owns the JDBC modules whereas the Administrator only deploys the module.

This means that the Administrator has limited control over packaged modules. When deploying a resource module, an Administrator can change the specified resource properties in the module, but the Administrator cannot add or delete modules. (As with other Jakarta EE modules, deployment configuration changes for a resource module are stored in a deployment plan for the module, leaving the original module untouched.)

- Standard Jakarta EE Application Modules
- Proprietary JDBC Application Modules

Standard Jakarta EE Application Modules

Jakarta EE 8 provides the option to programmatically define `DataSource` resources as application modules for a more flexible and portable method of database connectivity. See Using DataSource Resource Definitions in *Developing JDBC Applications for Oracle WebLogic Server*.

Proprietary JDBC Application Modules

JDBC resources can also be managed as application modules, similar to standard Jakarta EE modules. A proprietary JDBC application module is simply an XML file that conforms to the `jdbc-data-source.xsd` schema (available at `http://www.oracle.com/webfolder/technetwork/weblogic/jdbc-data-source/index.html`) and represents a 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 data source reconfiguration.

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 Oracle WebLogic Server*.

A proprietary JDBC application module can also be deployed as a stand-alone resource using the `weblogic.Deployer` utility or the WebLogic Remote 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-alone modules* and can be reconfigured using the WebLogic Remote 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 data source configuration between domains, such as between the development domain and the staging domain.

Note:

When deploying proprietary JDBC modules as standalone modules, a Multi Data Source needs to have a deployment order that is greater than the deployment orders of its member Generic data sources.

For more information about JDBC application modules, see Configuring JDBC Application Modules for Deployment.

For information about deploying stand-alone JDBC modules, see Deploying JDBC, JMS, and WLDF Application Modules in *Deploying Applications to Oracle WebLogic Server*.

- Including Drivers in EAR/WAR Files

Including Drivers in EAR/WAR Files

You can include a database driver in the `APP-INF/lib` directory of the EAR/WAR file that contains a packaged data source. This allows you to deploy a self-contained EAR/WAR file that has both the data source and driver required for an application.

Note:

You do not need to update the `classpath` of the manifest file to include the driver location.

An EAR has its own classloader and it is shared across all of the nested applications so any of them can use it. You can deploy multiple EAR/WAR files, each with a different driver version. However, if there are other versions of the driver in the system classpath, set `prefer-web-inf-classes=`*`true`* in the `weblogic.xml` file to ensure that the application uses the driver classes that it was packaged with which it was packaged.

You can also use the `WEB-INF/lib` directory to hold driver JAR files. The following example shows the location of the various directories in WAR and EAR files.

```
Application (ear)
    Web module (war)
        WEB-INF/lib
```

```
EJB module
META-INF
APP-INF/lib
```

However, you cannot have two versions of the same JAR in both `DOMAIN_HOME/lib` (see Using a Third-Party JAR File in DOMAIN_HOME/lib or the system classpath and `WEB-INF/lib` or `APP-INF/lib`, with `prefer-web-inf-classes` or `prefer-application-packages` set. That is, you should do only one of the following:

- Use `DOMAIN_HOME/lib` or system `classpath` to get the driver into all applications in the domain.
- Use the driver embedded in the application.

Note:

If you do not adhere to this restriction, it is possible (depending on the JAR, the version changes, and the order in which the JARs are referenced) that a `ClassCastException` will occur in the application.

If the JAR files are present in multiple locations, the following rules apply:

- If `prefer-web-inf-classes` in the `weblogic.xml` is false, the precedence is: `system classpath` > `DOMAIN_HOME/libAPP-INF/libWEB-INF/lib`.
- If `prefer-web-inf-classes` in `weblogic.xml` is true, the classes in `WEB-INF/lib` will take precedence over all other locations.

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

WebLogic Server identifies the data source defined in the JDBC module with a specific name.

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 run-time 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 and Using Production Redeployment to Update Applications in *Deploying Applications to Oracle WebLogic Server*.

JDBC Schema

In support of the modular deployment model for JDBC resources in WebLogic Server, Oracle 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.oracle.com/webfolder/technetwork/weblogic/jdbc-data-source/index.html`.

JDBC Data Source Type

Data sources should have a `datasource-type` set in the descriptor. This functionality was added in WebLogic Server 12.2.1 and is optional for backward compatibility.

The valid values are:

- `Generic`—Generic data source
- `MDS` —Multi Data Source
- `AGL`—Active GridLink data source
- `UCP`—Universal Connection Pool data source

If the `datasource-type` is not set to `UCP` the following validations are performed:

- If `datasource-type` is set to `AGL`, it is treated as an Active GridLink data source even if `FAN` enabled is `false` and no `ONS` list is configured, and the Active GridLink flag is `false`.
- If the `datasource-type` is not set to `AGL`, it is an error even if `FAN` enabled is true or an `ONS` list is configured or the Active GridLink flag is `true`.
- If no data source list exists (it does not have Multi Data Source members) and `datasource-type` is set to anything other than `GENERIC` or `AGL`, it is an error.
- If the data source list exists (it has Multi Data Source members) and the `datasource-type` is set to anything other than `MDS`, it is an error.

JMX and WLST Access for JDBC Resources

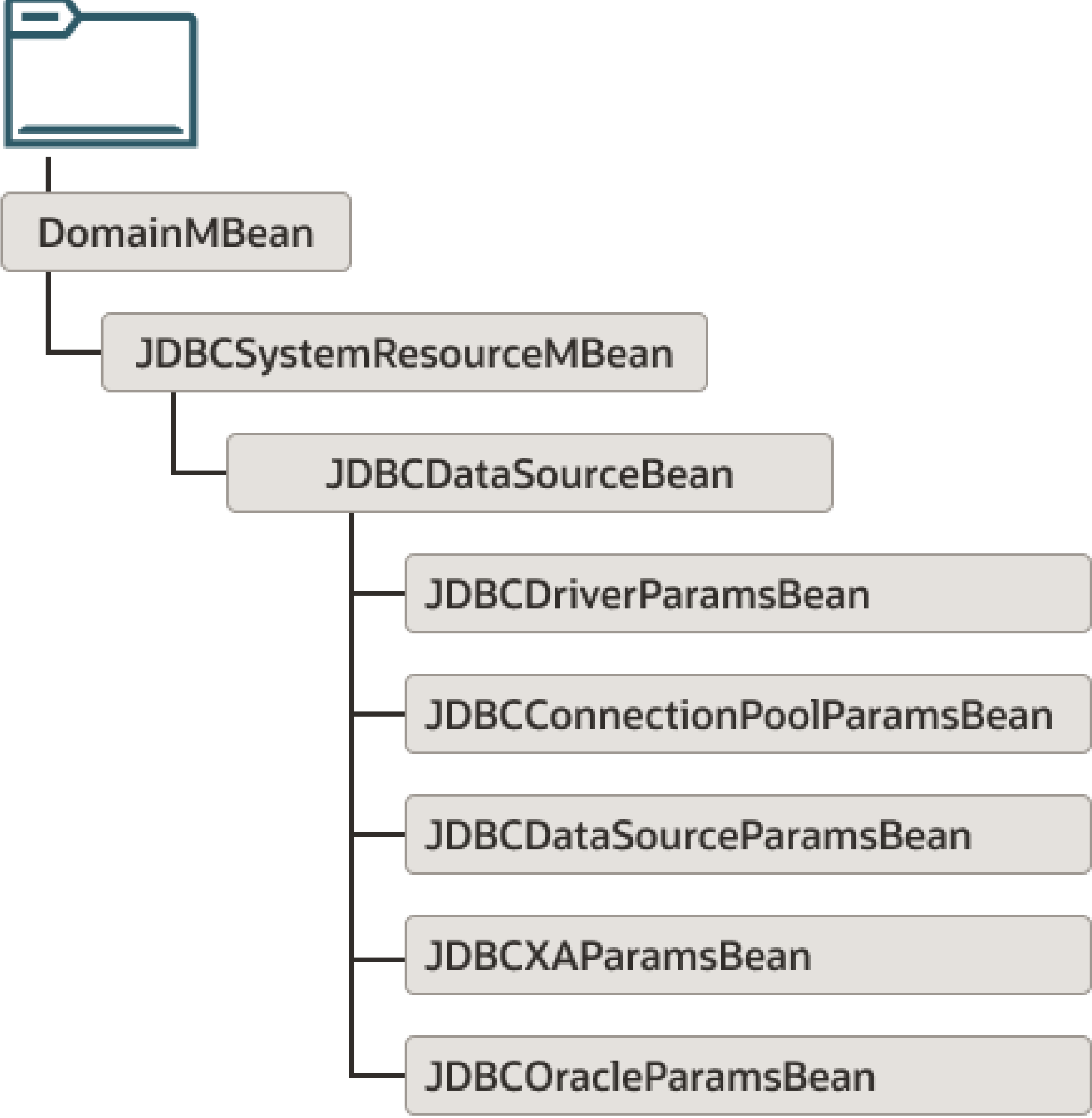
This section describes how to access WebLogic Server MBeans using JMX client or the WebLogic Scripting Tool (WLST).

- JDBC MBeans for System Resources
- JDBC Management Objects in the Jakarta Management Model (the Jakarta Management Specification Support)
- Using WLST to Create JDBC System Resources
- How to Modify and Monitor JDBC Resources
- Best Practices when Using WLST to Configure JDBC Resources

JDBC MBeans for System Resources

Figure 2-3 shows the hierarchy of the MBeans for JDBC objects in a WebLogic domain.

Figure 2-3 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 Jakarta Management Model (the Jakarta Management Specification Support)

The WebLogic Server JDBC subsystem supports the Jakarta Management Specification, which defines the Jakarta Management Model. The Jakarta Management Model is used for monitoring the run-time state of a Jakarta EE application server and its resources. You can access the Jakarta Management Model to monitor resources, including the WebLogic JDBC subsystem as a whole, JDBC drivers loaded into memory, and JDBC data sources.

To comply with the specification, Oracle added the following run-time MBean types for the WebLogic JDBC subsystem:

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

Note:

WebLogic JDBC run-time MBeans do not implement the optional Statistics Provider interfaces specified by the Jakarta Management Specification.

For more information about using the Jakarta management model with WebLogic Server, see *Developing Jakarta Management Applications for Oracle WebLogic Server*.

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, data sources, Multi Data Sources, and JDBC drivers.
- Target your JDBC system module.

Example 2-1 WLST Script to Create JDBC Resources

```
#----------------------------------------------------------------------
# Create JDBC Resources
#----------------------------------------------------------------------

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 resources 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")))

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

# 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 run-time objects using get methods.

See Navigating MBeans (WLST Online) in *Understanding the WebLogic Scripting Tool*.

Best Practices when Using WLST to Configure JDBC Resources

- Trap for Null MBean objects (such as pools, data sources, and drivers) before trying to manipulate the MBean object.
- When using WLST offline, the following characters are not valid in names of management objects: period (.), forward slash (/), or backward slash (\). See Syntax for WLST Commands in *Understanding the WebLogic Scripting Tool*.

Creating High-Availability JDBC Resources

To improve the availability of your JDBC resource and load balance communication between resources you can target or deploy a JDBC data source to the members of a cluster using the WebLogic Remote Console.

However, connections do not failover in the event that a cluster member becomes unavailable for any reason. New connections are created as needed on available cluster members. See Deploying Data Sources on Servers and Clusters.

Note:

A Multi Data Source can only use Generic data sources that are deployed on the same cluster member (in the same JVM).

3 Configure Database Connectivity

In WebLogic Server, you configure database connectivity through JDBC data sources, either in your WebLogic domain configuration or in your enterprise application.

- Using JDBC Drivers with WebLogic Server
- Using Oracle JDBC Driver Extensions
  In 23ai and following versions, the Oracle JDBC Driver can be extended through a Java Service Provider Interface (SPI). The SPI provides the URL, user, password, and JDBC parameters from an external source.
- JDBC Diagnosability
  In 23ai, diagnosability and logging are enhanced to improve the user experience.
- Configuring JDBC Data Sources

Using JDBC Drivers with WebLogic Server

WebLogic Server uses JDBC drivers to provide access to various databases. WebLogic Server comes with a default set of JDBC drivers but third-party JDBC drivers can also be used.

- Types of JDBC Drivers
  JDBC drivers listed in the WebLogic Remote Console when creating a data source are not necessarily certified for use with WebLogic Server. JDBC drivers are listed as a convenience to help you create connections to many of the database management systems available.
- JDBC Driver Support
  WebLogic Server provides support for application data access to any database using a JDBC-compliant driver.
- JDBC Drivers Installed with WebLogic Server
  The Oracle JDBC Thin driver 23ai is installed with Oracle WebLogic Server 14.1.2.0.0. In addition to the Oracle Thin Driver, the mySQL Connector/J 8.0 (`mysql-connector-j-8.2.0.jar`) JDBC driver, WebLogic-branded DataDirect drivers are also installed with WebLogic Server.
- Adding Third-Party JDBC Drivers Not Installed with WebLogic Server
- Globalization Support for the Oracle Thin Driver

Types of JDBC Drivers

JDBC drivers listed in the WebLogic Remote Console when creating a data source are not necessarily certified for use with WebLogic Server. 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 WebLogic Remote 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 Adding Third-Party JDBC Drivers Not Installed with WebLogic Server.

When a JDBC driver is updated, configuration requirements may change. The WebLogic Remote Console uses 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.

WebLogic Server provides the following JDBC drivers:

- **Oracle Thin Drivers**
  - Oracle Thin Driver XA
  - Oracle Thin Driver non-XA

  The following table lists nine Oracle Thin Drivers as they appear in WebLogic Remote Console, a sample of the URL format that is generated from the input provided by the user, and the class name of the driver configured:

Oracle Drivers	URL Format	Description	Driver Class Name
Oracle's Driver (Thin XA) for Application Continuity; Versions: Any	`jdbc:oracle:thin:@hostname:port/service`	Database is used as service. The service should be available on a single instance for Generic and Multi Data Source.	`oracle.jdbc.replay.OracleXADataSourceImpl`
Oracle's Driver (Thin XA) for Instance connections; Versions: Any	`jdbc:oracle:thin:@hostname:port:SID`	Database is used as SID, the use of SID is deprecated. Use service name instead of SID in this format.	`oracle.jdbc.xa.client.OracleXADataSource`
Oracle's Driver (Thin XA) for RAC Service-Instance connections; Versions: Any	`jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=hostname)(PORT=hostname)))(CONNECT_DATA=(SERVICE_NAME=service)(INSTANCE_NAME=instance)))`	Use this format when the service is available on multiple instances and the URL should map to a single instance for Generic and Multi Data Source. A long format URL is generated so that you can specify instance name.	`oracle.jdbc.xa.client.OracleXADataSource`
Oracle's Driver (Thin XA) for Service connections; Versions: Any	`jdbc:oracle:thin:@//hostname:port/service`	Database is used as service. The service should be available on a single instance for Generic and Multi Data Source.	`oracle.jdbc.xa.client.OracleXADataSource`
Oracle's Driver (Thin XA) for TNS Alias, Driver Extension Application Continuity Connections Versions:Any	`jdbc:oracle:thin:@tnsalias`	Database is used as the tns alias. When specified the driver property oracle.net.tns_admin is set to the location of tnsnames.ora, Oracle wallet or keystore.	`oracle.jdbc.replay.OracleXADataSourceImpl`

Oracle Drivers	URL Format	Description	Driver Class Name
Oracle's Driver (Thin XA) for TNS Alias, Driver Extension Connections Versions:Any	`jdbc:oracle:thin:@`*`tns alias`*	Database is used as the tns alias. When specified the driver property oracle.net.tns_admin is set to the location of tnsnames.ora, Oracle wallet or keystore.	`oracle.jdbc.xa.client.OracleXADataSource`
Oracle's Driver (Thin) for Application Continuity; Versions: Any	`jdbc:oracle:thin:@//`*`hostname:port/service`*	Database is used as service. The service should be available on a single instance for Generic and Multi Data Source.	`oracle.jdbc.replay.OracleDataSourceImpl`
Oracle's Driver (Thin) for Consolidated AC/TAC; Versions:21+	`jdbc:oracle:thin:@//`*`hostname:port/service`*	Database is used as service. The driver class supports application continuity, is a pooled data source and is only available in Oracle JDBC 21 or later versions.	`oracle.jdbc.datasource.impl.OracleDataSource`
Oracle's Driver (Thin) for Instance connections; Versions: Any	`jdbc:oracle:thin:@`*`hostname:port:SID`*	Database is used as SID, the use of SID is deprecated. Use the service name instead of SID in this format.	`oracle.jdbc.OracleDriver`
Oracle's Driver (Thin) for pooled instance connections; Versions: Any	`jdbc:oracle:thin:@`*`hostname:port:SID`*	Database is used as SID. Use this format to get a pooled data source, this is not a very commonly used format.	`oracle.jdbc.pool.OracleDataSource`
Oracle's Driver (Thin) for RAC Service-Instance connections; Versions: Any	`jdbc:oracle:thin:@`*`(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=hostname)(PORT=port)))(CONNECT_DATA=(SERVICE_NAME=service)(INSTANCE_NAME=instance))`*	Use this format when the service is available on multiple instances and the URL should map to a single instance for Generic and Multi Data Source. A long format URL is generated so that you can specify instance name.	`oracle.jdbc.OracleDriver`
Oracle's Driver (Thin) for Service connections; Versions: Any	`jdbc:oracle:thin:@//`*`hostname:port/service`*	Database is used as service. The service should be available on a single instance for Generic and Multi Data Source.	`oracle.jdbc.OracleDriver`

Oracle Drivers	URL Format	Description	Driver Class Name
Oracle's Driver (Thin) for TNS Alias Connections Versions:Any	`jdbc:oracle:thin:@tns alias`	Database is used as the tns alias. When specified the driver property oracle.net.tns_admin is set to the location of tnsnames.ora, Oracle wallet or keystore.	`oracle.jdbc.OracleDriver`
Oracle's Driver (Thin) for TNS Alias, Driver Extension Application Continuity Connections Versions:Any	`jdbc:oracle:thin:@tns alias`	Database is used as the tns alias. When specified the driver property oracle.net.tns_admin is set to the location of tnsnames.ora, Oracle wallet or keystore.	`oracle.jdbc.replay.OracleDataSourceImpl`
Oracle's Driver (Thin) for TNS Alias, Driver Extension Consolidated AC/TAC Connections Versions:21+	`jdbc:oracle:thin:@tns alias`	Database is used as the tns alias. When specified the driver property oracle.net.tns_admin is set to the location of tnsnames.ora, Oracle wallet or keystore. The driver class supports application continuity, is a pooled data source and is only available in Oracle JDBC 21 or later versions.	`oracle.jdbc.datasource.impl.OracleDataSource`

- **MySQL (non-XA)**
- **Third-party JDBC drivers**

  For more information, see Using JDBC Drivers with WebLogic Server.
- **WebLogic-branded DataDirect drivers**: These drivers are available for the following database management systems:
  - DB2
  - Microsoft SQL Server

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`.

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 Configurations in *What's New in Oracle WebLogic Server*.

JDBC Driver Support

WebLogic Server provides support for application data access to any database using a JDBC-compliant driver.

The JDBC-compliant driver needs to meet the following requirements:

- The driver must be thread-safe.
- The driver must implement standard JDBC transactional calls, such as `setAutoCommit()` and `setTransactionIsolation()`, when used in transactional aware environments.
- If the driver that does not implement serializable or remote interfaces, it cannot pass objects to an RMI client application.

When WebLogic Server features use a database for internal data storage, database support is more restrictive than for application data access. The following WebLogic Server features require internal data storage:

- Container Managed Persistence (CMP)
- Rowsets
- JMS/JDBC Persistence and use of a WebLogic JDBC Store
- JDBC Session Persistence
- RDBMS Security Providers
- Database Leasing (for singleton services and server migration)
- JTA Logging Last Resource (LLR) optimization.

JDBC Drivers Installed with WebLogic Server

The Oracle JDBC Thin driver 23ai is installed with Oracle WebLogic Server 14.1.2.0.0. In addition to the Oracle Thin Driver, the mySQL Connector/J 8.0 (`mysql-connector-j-8.2.0.jar`) JDBC driver, WebLogic-branded DataDirect drivers are also installed with WebLogic Server.

The drivers files are named `ojdbc11.jar` for JDK17 and JDK21.

Note:

See Using WebLogic-branded DataDirect Drivers in *Developing JDBC Applications for Oracle WebLogic Server*.

These drivers are installed in subdirectories of `$ORACLE_HOME/oracle_common/modules`. The manifest in the `weblogic.jar` lists this file so that it is loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add this JDBC driver 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 Configurations in *What's New in Oracle WebLogic Server* .

> **Note:**
>
> WebLogic Server includes a version of the Derby DBMS installed with the WebLogic Server examples in the `WL_HOME\common\derby` directory. Derby is an all-Java DBMS product included in the WebLogic Server distribution solely in support of demonstrating the WebLogic Server examples. For more information about Derby, see `http://db.apache.org/derby`.

Adding Third-Party JDBC Drivers Not Installed with WebLogic Server

To use third-party JDBC drivers that are not installed with WebLogic Server, you can add them to the `DOMAIN_HOME/lib` directory.Here, `DOMAIN_HOME` represents the directory in which the WebLogic Server domain is configured. The default path is `ORACLE_HOME/user_projects/domains`.
For more information, see Adding JARs to the Domain /lib Directory in *Developing Applications for Oracle WebLogic Server*.

> **Note:**
>
> In previous releases, adding a new JDBC driver or updating a JDBC driver where the replacement JAR has a different name than the original JAR required updating the WebLogic Server's classpath to include the location of the JDBC driver classes. This is no longer required.

Using a Third-Party JAR File in DOMAIN_HOME/lib

Using a third-party JAR file in `DOMAIN_HOME/lib` is only supported for third-party JDBC drivers that are not installed with WebLogic Server. The drivers installed with WebLogic Server are described in JDBC Drivers Installed with WebLogic Server.

When you use a third-party JAR file in the `DOMAIN_HOME/lib` directory, note the following:

- The classloader that gets created is a child of the system classpath classloader in WebLogic Server.
- Any classes that are in JARs in this directory are visible only to Jakarta EE applications in the server, such as EAR files.
- You can use the WebLogic Remote Console and WLST online to configure and manage the JAR files. (You may also be able to use WLST offline because the data source is not deployed.)
- These JAR files do not work when run from a standalone client (such as the t3 RMI client) or standalone applications (such as `java utils.Schema`).
- If there are multiple domain directories involved (that is, multiple machines without a shared file system), the JAR file must be installed in `/lib` in each domain directory.
- WebLogic Server use of methods called on third-party drivers (such as TimesTen `abort` and `DB2 setDB2ClientUser`) is supported.

Note:

For details on WebLogic Server functionality supported with these JAR files, see Database Interoperability in *What's New in Oracle WebLogic Server*, and the appropriate version of the Oracle Fusion Middleware Supported System Configurations matrix documentation for specific database driver and DB version certification information.

Data Source Support

Third-party JAR files installed `in /lib` can be used with the following:

- All data source types supported by WebLogic Server system resources including Generic, Multi Data Source, and Active GridLink. The Universal Connection Pool data source does not apply since the UCP JAR is not third-party.
- Packaged data sources in an EAR or a WAR.
- Jakarta EE 8 data source definition defined in an EAR or WAR.

Although not JDBC methods, using a third-party JAR file `in /lib` does apply to WebLogic Server data source callbacks like Multi Data Source failover, connection, replay, and harvesting.

Example 3-1 Example of Using a Third-Party JAR File in /lib

The following example shows the files contained in a standalone WAR file named `getversion.war`. The Derby JAR files are located in WEB-`INF/lib` or `DOMAIN_HOME/lib` (or both). The class file is compiled and installed at `WEB-INF/classes/demo/GetVersion.class`.

```
<web-app>
  <welcome-file-list>
    <welcome-file>welcome.jsp</welcome-file>
  </welcome-file-list>
  <display-name>GetVersion</display-name>
  <servlet>
    <description></description>
    <display-name>GetVersion</display-name>
    <servlet-name>GetVersion</servlet-name>
    <servlet-class>
       demo.GetVersion
    </servlet-class>
  </servlet>
<!-- Data source description can go in the web.xml descriptor or as an
annotation in the java code - see below
  <data-source>
    <name>java:global/DSD</name>
    <class-name>org.apache.derby.jdbc.ClientDataSource</class-name>
    <port-number>1527</port-number>
    <server-name>localhost</server-name>
    <database-name>examples</database-name>
    <transactional>false</transactional>
  </data-source>
-->
</web-app>
```

```
WEB-INF/weblogic.xml

<weblogic-web-app>
  <container-descriptor>
    <prefer-web-inf-classes>true</prefer-web-inf-classes>
  </container-descriptor>
</weblogic-web-app>

Java file

package demo;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.annotation.sql.DataSourceDefinition;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

@DataSourceDefinition(name="java:global/DSD",
className="org.apache.derby.jdbc.ClientDataSource",
portNumber=1527,
serverName="localhost",
databaseName="examples",
transactional=false
)
@WebServlet(urlPatterns = "/GetVersion")
public class GetVersion extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {
  @Resource(lookup = "java:global/DSD")
  private DataSource ds;

  public GetVersion() {
    super();
  }

  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    doPost(request, response);
  }

  protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");

    PrintWriter writer = response.getWriter();
    writer.println("<html>");
    writer.println("<head><title>GetVersion</title></head>");
    writer.println("<body>" + doit() +"</body>");
    writer.println("</html>");
    writer.close();
```

```
    }

    private String doit() {
      String ret = "FAILED";
      Connection conn = null;
      try {
        conn = ds.getConnection();
        ret = "Connection obtained with version= " +
          conn.getMetaData().getDriverVersion();
      } catch(Exception e) {
        e.printStackTrace();
      } finally {
        try {
          if (conn != null)
            conn.close();
        } catch (Exception ignore) {}
      }
      return ret;
    }
  }
```

Globalization Support for the Oracle Thin Driver

For globalization support with the Oracle Thin driver, Oracle supplies the `orai18n.jar` file.This file 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` and `orai18n-mapping.jar` in your `CLASSPATH`.

The `orai18n.jar` and `orai18n-mapping.jar` are included with the WebLogic Server installation in the *`ORACLE_HOME`*`\oracle_common\modules\oracle.nlsrtl` folder. These files are *not* referenced by the `weblogic.jar manifest` file, so you must add them to your `CLASSPATH` before they can be used.

Using Oracle JDBC Driver Extensions

In 23ai and following versions, the Oracle JDBC Driver can be extended through a Java Service Provider Interface (SPI). The SPI provides the URL, user, password, and JDBC parameters from an external source.

There is an open-source project at open-source project for “Oracle JDBC Driver Extensions” that provides some already-written “providers” that use the SPI to read the parameter from external vaults. There are providers for OCI vault, Azure vault, and Open Telemetry.

Using the vault enables you to remove the credential information from a local schema file for security reasons and store the information in a single location for all applications that use the database. The benefits of using JDBC Extensions is that it centralizes where the database information can be updated with no change to the application or to the Data Source configuration in WebLogic Server. This is beneficial when the application uses many data sources.

- Using OCI Vault
- Storing Wallet and DB Connection String

Using OCI Vault

You need to setup an OCI account to use the OCI vault. Perform the following steps to use the OCI vault:

Setting Up an OCI Account

1. Open the following URL.

   https://www.oracle.com/cloud/sign-in.html

2. Enter your details as listed below:
   - Country/territory
   - First name
   - Last name
   - Email
3. Verify your email address.
4. Select **Corporate/Individual**.

   This option is for a work-related account.

5. Select a Home Region.

   Ensure you pick the closest location to avoid any network delay.

6. Set up a password meeting the criteria.
7. Set up 2FA using the Oracle Authenticator.

   After setting up your OCI account, you must setup configuration for access to your client machine by installing the OCI Command Line Interface (CLI). For more information, see Quickstart.

   For example,

   Run the following command on OL8:

   ```
   sudo dnf -y install oraclelinux-developer-release-el8


   sudo dnf install python36-oci-cli
   ```

Creating Configuration File

8. Run the following command:

   ```
   oci setup config
   ```

9. Use the following default value as the file name:

   ```
   ~/.oci/config
   ```

10. In the **OCI** dashboard, click **Profile** in the top left corner.
11. Copy the OCID.

    This is the user OCID.

12. Click **Profile**, select **Tenancy** and copy the OCID.

This is the Tenancy OCID.

13. Select the region associated with the account.

    A list of regions and associated numbers are available.

14. Create a new public/private key pair.

    The configuration file is created.

15. Follow the steps in How to Upload the Public Key to upload the public key.

Storing Wallet and DB Connection String

Oracle JDBC Extension stores a URL string, username, encrypted password, and optional JDBC parameters in an OCI secret within an OCI vault. It works on any Oracle database and not only a cloud database. You can use an existing database or create a new OCI database.

Perform the following steps to create a wallet and store it:

1. Create the database.
2. Use SQL to create a user with a password and grant **CREATE SESSION** privilege.
3. Click on Database Connection, lookup for the <databasename>_tp TNS name, and click on copy to get the URL.

   You will need the URL to create the secret.

> **Note:**
>
> There is no mechanized way to create the secret. You do not need to have the wallet and TNS_ADMIN set on the local client.
>
> It is necessary to create a wallet for storing it in a secret.

Create a Vault

4. In the OCI dashboard, select the menu in the top left corner.
5. Select **Identity & Security** > **Vault** > **Create Vault**.
6. Enter a name and create vault.
7. Click on the vault name to enter the vault.
8. Create a **Master Encryption Key** by selecting **Create Key**.
9. Enter a name and click **Create Key**.

Create a Secret for DB Wallet

10. In the left margin, click **Secrets**.
11. Click **Create Secret** and enter a name.
12. Click **Manual Secret Generation** and paste the password enclosed in double quotes into the text box.
13. Click **Secret**.
14. Click the name of the secret.
15. Click **Copy for the OCID**.
16. Create and download database wallet.

17. After unzipping the wallet, run the following command to get the base-64 string for the wallet:

```
base64 cwallet.sso > output
```

18. Edit the output to ensure that no spaces are there in a single string.
19. To create a secret with this string, click **Secrets** > **Create Secret**.
20. Enter a name and click **Manual Secret Generation**.

> **Note:**
>
> It is critical to change the Secret Type Template to Base64

21. Copy and paste the base64 output from the `cwallet.sso` into the text box.

Create a Secret for DB Password

22. In the left pane, click **Secrets**.
23. Click **Create Secret** and enter a name.
24. Click **Manual Secret Generation**.
25. Select **Secret Type Template (Select Plain-Text to enter plain text password)**.
26. Enter DB Password in **Secret Contents**.
27. Click **Create Secret**.
28. Click the name of the secret and save the OCID of DB Password.
29. Create a Secret for DB Connection Details.
30. Click **Create Secret**.
31. Click the name of the secret and save the OCID.

    You can now create your secret that will be referenced by the driver.
32. In the left margin, click **Secrets**.
33. Click **Create Secret** and enter a name.
34. Click **Manual Secret Generation**.

    You need to manually create the text and it should be done out of the OCI console. For information on the format, see https://docs.oracle.com/en/database/oracle/oracle-database/23/jajdb/oracle/jdbc/spi/OracleConfigurationProvider.html.

    Given below is an example where the host and service name come from the TNS name that you saved from the datasource, the username is the user that you created in the database, the secrets for the password and `wallet_location` that you created in the vault.

```
{"connect_descriptor":
"(description=(retry_count=20)(retry_delay=3)(address=(protocol=tcps)
(port=1522)(host=dbhost.oraclecloud.com))
(connect_data=(service_name=dbservice.oraclecloud.com))
(security=(ssl_server_dn_match=yes)))",
"user": "DEMO1",
"password": {
     "type": "ocivault",
```

```
     "value": "ocid1.vaultsecret.oc1.ca-toronto-
1.amaaaaaadngsnoaajxsjica4vqggkcq2f4jxpillfc6nv3x55fs7btedglka"
},
"wallet_location": {
     "type": "ocivault",
     "value": "ocid1.vaultsecret.oc1.ca-toronto-
1.amaaaaaadngsnoaaftfj2ll37ex67wru3wqhqsv2q6x2x67epegzl2gvkm6a"

},
"jdbc": {
  "oracle.jdbc.ReadTimeout": 1006,
  "defaultRowPrefetch": 20
}
}
```

35. Copy and paste the Json string into the **Secret Contents** text box for the manual secret.
36. Click **Create Secret**.
37. Click the name of secret that you just created.
38. Click Copy next to the OCID.

    You must use the OCID of the secret to generate the URL of the form `jdbc:oracle:thin:@config-ocivault://OCID-FOR-SECRET`.

    For example,

    ```
    jdbc:oracle:thin:@config-ocivault://ocid1.vaultsecret.oc1.ca-toronto-
    1.amaaaaaadngsnoaabnyfwwxnmz67flenjpy26ikuylyui45n6p3xfx76h3dq
    ```

    If you need to change the secret value, you must click the name of the secret. Click **Create Secret Version** to create a new version, enter the text, and click **Create Secret Version**.

Note:

When you create a new secret version, it does not change the OCID so you do not need to update the OCID wherever it is used.

JDBC Diagnosability

In 23ai, diagnosability and logging are enhanced to improve the user experience.

In previous releases, the debug JAR files are used for logging purposes. These files are indicated with a `_g` in the file name. For example, `ojdbc8_g.jar` or `ojdbc11_g.jar` and must be included in the CLASSPATH. The enhanced diagnosability feature eliminates the need to use the debug JAR files and also the need to switch between the regular JAR files and the debug JAR files.

- Enabling JDBC Driver Logging
- Diagnosing First Failure

Enabling JDBC Driver Logging

Perform the following steps to enable logging and diagnose failures:

1. Set the value of the following system properties to true:

```
-Doracle.jdbc.diagnostic.enableLogging=true
```

```
-Doracle.jdbc.diagnostic.enableSensitiveDiagnostics=true
```

```
-Doracle.jdbc.diagnostic.permitSensitiveDiagnostics=true
```

2. Configure java util logging with `-Djava.util.logging.config.file=./logging.config` using the following logging file:

```
oracle.jdbc.handlers =
java.util.logging.FileHandleroracle.jdbc.level =
FINESTjava.util.logging.FileHandler.level =
FINESTjava.util.logging.FileHandler.pattern =
%h/jdbc_incident_%u.logjava.util.logging.FileHandler.limit =
50000java.util.logging.FileHandler.count =
10java.util.logging.FileHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
```

   or

```
handlers = java.util.logging.ConsoleHandler
oracle.jdbc.level = FINEST
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
```

3. Set the value of the following system property to true:

```
-Doracle.ucp.diagnostic.enableLogging=true
```

Diagnosing First Failure

The Diagnose First Failure feature is one of the major features of the enhanced diagnosability.

When this feature is enabled, it diagnosis the first occurrence of a failure and the most critical trace information is captured in the memory. After that memory fills, the oldest trace records are overwritten and are subsequently dumped into the `java.util.logging` when signaled. One of the following can signal the dumping of the trace records:

```
Occurrence of an error in the connection Client application code calling an API
An MBean
```

This feature captures only the most critical information which includes the extent of information that provides a reasonable chance of diagnosing the most likely problems. In the public mode, this information is severely restricted so that any sensitive information is omitted. In the sensitive mode, this information includes the entire network conversation.

This feature is enabled by default. You can disable it either by setting the `CONNECTION_PROPERTY_ENABLE_DIAGNOSE_FIRST_FAILURE` property to `FALSE` or through the DiagnosticMBeans interface.

JDBC diagnosability feature is enhanced in this release to make it more user-friendly.

Configuring JDBC Data Sources

In WebLogic Server, you configure database connectivity by adding JDBC data sources to your WebLogic domain. Configuring data sources requires several steps including choosing a type of data source, creating the data source, configuring connection pools and Oracle database parameters and so on.

- Creating a JDBC Data Source
  WebLogic JDBC data sources provide database access and database connection management.
- Configuring Connection Pool Features
- Advanced Connection Properties
  You can set up advanced connection properties like fatal error codes and use of Edition-Based Redefinition (EBR). You define fatal error codes which indicate the database server with which the data source communicates is no longer accessible on a connection. EBR provides the ability to upgrade the database component of an application while it is in use, thereby minimizing or eliminating down time.
- Configure Oracle Parameters
  WebLogic Server provides several attributes that provide improved data source performance when using Oracle drivers.
- Configure ONS Client Parameters
- Tuning Generic Data Source Connection Pools
- Generic Data Source Handling for Oracle RAC Outages
- Generic Data Source Handling of Driver-Level Failover

Creating a JDBC Data Source

WebLogic JDBC data sources provide database access and database connection management.

You can create and manage JDBC data sources using the following management tools:

- **Oracle WebLogic Remote Console**: See Data Sources in *Oracle WebLogic Remote Console Online Help*.
- **WebLogic Scripting Tool (WLST)**: See WLST Online Sample Scripts in *Understanding the WebLogic Scripting Tool*.

  Example:

  ```
  EXAMPLES_HOME\wl_server\examples\src\examples\wlst\online\jdbc_data_source_
  creation.py
  ```

  where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured.

You can configure data sources in the WebLogic Remote Console.

- Configure JDBC Data Source Properties
  JDBC data source properties include options that determine the identity of the data source and the way database connection handles the data.

- Configure Transaction Options
  When you configure a JDBC data source using the WebLogic Remote Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver. XA, non-XA, and Global transaction options are supported by WebLogic JDBC data sources.
- Configure Connection Properties
  Connection Properties allows you to configure the connection between the data source and the DBMS. Typical attributes are the database name, host name, port number, user name, and password.
- Configure Testing Options
  Test Database Connection allows you to test a database connection before the data source configuration is finalized using a table name or SQL statement.
- Target JDBC Data Sources
  You can select one or more targets to which to deploy your new JDBC data source. If you don't select a target, the data source will be created but not deployed. You will need to deploy the data source at a later time before getting connections.

Configure JDBC Data Source Properties

JDBC data source properties include options that determine the identity of the data source and the way database connection handles the data.

Data Source Names: You can use JDBCA data source name to identify the data source within the WebLogic domain. For system resource data sources, names must be unique among all other JDBC system resources. To avoid naming conflicts, data source names should also be unique among other configuration object names, such as servers, applications, clusters, and JMS queues, topics, and servers. For JDBC application modules packaged in an application, data source names must be unique among JDBC data sources with a similar scope.

The data source name cannot contain the following special characters: `@ # $`.

Data Source Scope: You can select the scope for the data source and set the scope to Global (at the domain level). See *Target Data Sources*.

JNDI Names: You can configure a data source so that it binds to the JNDI tree with a single or multiple names. See *Developing JNDI Applications for Oracle WebLogic Server*.

Database Type: You can select the Database Management System (DBMS) of the database you want to connect. For information about supported databases, see Supported Configurations in *What's New in Oracle WebLogic Server*.

JDBC Driver: You can select a JDBC database driver that is preferred to create a database connection. You should verify, however, that the URL is as you want it before asking the console to test it. 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 WebLogic Remote Console are shipped (and/or are already in the `classpath`) with WebLogic Server. See Types of JDBC Drivers.

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`.

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 Configurations in *What's New in Oracle WebLogic Server*.

Configure Transaction Options

When you configure a JDBC data source using the WebLogic Remote Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver. XA, non-XA, and Global transaction options are supported by WebLogic JDBC data sources.

For more information on configuring transaction support for a data source, see JDBC Data Source Transaction Options.

Configure Connection Properties

Connection Properties allows you to configure the connection between the data source and the DBMS. Typical attributes are the database name, host name, port number, user name, and password.

> **Note:**
>
> You can use a Single Client Access Name (SCAN) address to represent the host name.
>
> - If you set the Oracle RAC `REMOTE_LISTENER` parameter for your data source to `SCAN`, then the data source connection URL can only use a SCAN address.
> - If you set the Oracle RAC `REMOTE_LISTENER` parameter for your data source to `List of Node VIPs`, then the data source connection URL can only use a list of VIP addresses.
> - If you set the Oracle RAC `REMOTE_LISTENER` parameter for your data source to `Mix of SCAN and List of Node VIPs`, then the data source connection URL can use both SCAN and VIP addresses.
>
> See Introduction to Oracle RAC in *Real Application Clusters Administration and Deployment Guide*.

Configuring Connection Properties for Oracle BI Server: If you selected Oracle BI Server as your DBMS, configure the additional connection properties on the Connection Properties page as described in Connection String in *Oracle Business Intelligence Publisher Administrator's and Developer's Guide*.

Configure Testing Options

Test Database Connection allows you to test a database connection before the data source configuration is finalized using a table name or SQL statement.

If necessary, you can test additional configuration information using the `Properties` and `System Properties` attributes.

Target JDBC Data Sources

You can select one or more targets to which to deploy your new JDBC data source. If you don't select a target, the data source will be created but not deployed. You will need to deploy the data source at a later time before getting connections.

For more information, see *Oracle WebLogic Remote Console Online Help* and Using JDBC Drivers with WebLogic Server.

Configuring 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.

Note:

If a non-dynamic data source attribute is updated, the data source needs to be undeployed or redeployed for the attribute to take effect. To determine whether an attribute is dynamic or non-dynamic, see the MBean reference MBean Reference for Oracle WebLogic Server for the attribute. If the attribute definition contains the `Redeploy or Restart required` text, then it is a non-dynamic attribute.

See *Oracle WebLogic Remote Console Online Help* and JDBCConnectionPoolParamsBean in *MBean Reference for Oracle WebLogic Server*.

Note:

Certain Oracle JDBC extensions, and possibly other non-standard methods available from other drivers may durably alter a connection's behavior in a way that future users of the pooled connection will inherit. WebLogic Server attempts to protect connections against some types of these calls when possible.

The following topics include information about connection pool options for a JDBC data source. Some of these options are dynamically changeable and others are non-dynamic.

- Enabling JDBC Driver-Level Features
- Enabling Connection-based System Properties
- Enabling Connection-based Encrypted Properties
- Initializing Database Connections with SQL Code

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 by 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.

Enabling Connection-based System Properties

WebLogic JDBC data sources support setting driver properties using the value of system properties. The value of each property is derived at runtime from the named system property. You can configure connection-based system properties using the WebLogic Remote Console by editing the `System Properties` attribute of your data source configuration.

If a system property value is set, it overrides an encrypted property value, which overrides a normal property value (you can only have one property value for each property name).

A system property value can contain one of the variables listed in Table 3-1. If one or more of these variables is included in the system property, it is substituted with the corresponding value. If a value is not found, no substitution is performed. If none of these variables are found in the system property, then the value is taken as a system property name.

Table 3-1 Variables Supported in System Property Values for JDBC Data Source

Variable	Value Description
`${pid}`	First half (up to @) of `ManagementFactory.getRuntimeMXBean().getName()`
`${machine}`	Second half of `ManagementFactory.getRuntimeMXBean().getName()`
`${user.name}`	Java system property `user.name`
`${os.name}`	System property `os.name`
`${datasourcename}`	Data source name from the JDBC descriptor. It does not contain the partition name.
`${partition}`	Partition name or DOMAIN
`${serverport}`	WebLogic Server server listen port
`${serversslport}`	WebLogic Server server SSL listen port
`${servername}`	WebLogic Server server name
`${domainname}`	WebLogic Server domain name

A sample set of properties is shown in the following example:

```
<properties>
<property>
  <name>user</name>
  <sys-prop-value>user</sys-prop-value>
</property>
<property>
  <name>v$session.osuser</name>
  <sys-prop-value>${user.name}</sys-prop-value>
</property>
<property>
  <name>v$session.process</name>
  <sys-prop-value>${pid}</sys-prop-value>
</property>
<property>
  <name>v$session.machine</name>
```

```
  <sys-prop-value>${machine}</sys-prop-value>
</property>
<property>
  <name>v$session.terminal</name>
  <sys-prop-value>${datasourcename}</sys-prop-value>
</property>
<property>
  <name>v$session.program</name>
  <sys-prop-value>WebLogic ${servername} Partition ${partition}</sys-prop-
value>
</property>
</properties>
```

In this example:

- user is set to the value of `-Duser=value`
- `v$session` values are set as described in Table 3-1

  For example, `v$session.program` running on `myserver` is set to `WebLogic myserver Partition DOMAIN`

Note that the values have the following length limitations:

- `osuser`—30
- `process`—24
- `machine`—64
- `terminal`—30
- `program`—48

Enabling Connection-based Encrypted Properties

WebLogic JDBC data sources support setting driver properties using encrypted values. You can configure connection-based encrypted properties using the WebLogic Remote Console by editing the `Encrypted Properties` attribute of your data source configuration. See Using Encrypted Connection Properties.

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 in the WebLogic Remote Console. Alternatively, you can specify simply a table name without `SQL` and the statement `SELECT COUNT(*) FROM tablename` is used. 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. An Oracle DBMS example:

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

The SQL statement is executed using `JDBC Statement.execute()`. Options that you can set using **InitSQL** vary by DBMS. See the documentation from your database vendor for supported statements. If you want to execute multiple statements, you may want to create a stored procedure and execute it. The syntax is vendor specific. For example, to execute an Oracle stored procedure:

```
SQL CALL MYPROCEDURE()
```

Advanced Connection Properties

You can set up advanced connection properties like fatal error codes and use of Edition-Based Redefinition (EBR). You define fatal error codes which indicate the database server with which the data source communicates is no longer accessible on a connection. EBR provides the ability to upgrade the database component of an application while it is in use, thereby minimizing or eliminating down time.

- Define Fatal Error Codes
- Using Edition-Based Redefinition

Define Fatal Error Codes

You can define fatal error codes that indicate that the database server with which the data source communicates is no longer accessible on a connection. The connection is marked invalid and taken out of the pool but the data source is not suspended. These errors include deployment errors that cause a server to fail to boot and connection errors that prevent a connection from being put back in the connection pool.

When specified as the exception code within a SQLException (retrieved by `sqlException.getErrorCode()`), it indicates that a fatal error has occurred, the connection is no longer good, and it is removed from the connection pool. For Oracle databases the following fatal error codes are predefined within WLS and do not need to be placed in the configuration file:

Error Code	Description
3113	end-of-file on communication channel
3114	not connected to ORACLE
1033	ORACLE initialization or shutdown in progress
1034	ORACLE not available
1089	immediate shutdown in progress - no operations are permitted
1090	shutdown in progress - connection is not permitted
17002	I/O exception

For DB2, the following fatal error codes are predefined: -4498, -4499, -1776, -30108, -30081, -30080, -6036, -1229, -1224, -1035, -1034, -1015, -924, -923, -906, -518, -514, 58004.

The fatal error codes feature can be disabled by defining the data source driver connection property `oracle.jdbc.OracleDriver` value as false.

Using Edition-Based Redefinition

Edition-based redefinition (EBR) provides the ability to upgrade the database component of an application while it is in use, thereby minimizing or eliminating down time. It allows a pre-

upgrade and post-upgrade view of the data to exist at the same time, providing a hot upgrade capability. You can then specify which view you want for a particular session.

See:

- Using Edition-Based Redefinition in *Oracle Database Development Guide*
- Edition-Based Redefinition White Paper at *Edition-Based Redefinition Technical Deep Dive*

Using EBR with JDBC Connections

There are two approaches to using EBR with JDBC connections:

- If you use a database service to connect to the database and an initial session edition was specified for that service, then the initial session edition for the service is your initial session edition on the connection. This approach is recommended for minimal overhead on the connection.

  When you create or modify a database service, you can specify its initial session edition. To create or modify a database service, Oracle recommends using the `srvctl add service` or `srvctl modify service` command. To specify the default initial session edition of the service, use the `-edition` option.

  Alternatively, you can create or modify a database service with the `DBMS_SERVICE.CREATE_SERVICE` or `DBMS_SERVICE.MODIFY_SERVICE` procedure, and specify the default initial session edition of the service with the EDITION attribute.
- Changing your session edition after connecting to the database using the SQL statement `ALTER SESSION SET EDITION`. You can change your session edition to any edition on which you have the USE privilege. Note that changing the edition can require re-generating a significant amount of state on session and database server. Oracle recommends using `DBMS_SESSION.RESET_PACKAGE` to clean-up some of this state when changing the edition on a session.

Using Edition-based redefinition does not require any new WebLogic Server functionality.

To make use of EBR, your environment needs to consist of an earlier version of the application with a data source that references the earlier EDITION and a later version of the application with a data source that references the later EDITION. When referring to multiple versions of a WebLogic Server application, you should be using WebLogic Server versioned applications in the production redeployment feature. See Developing Applications for Production Redeployment in *Developing Applications for Oracle WebLogic Server*. By combining Oracle database EBR and WebLogic Server versioned applications, the application can be upgraded with no downtime, making the combination of features more powerful than either feature independently.

You need to run with a versioned database and a versioned application initially so that you can switch versions. To version a WebLogic Server application, simply add the `Weblogic-Application-Version` property in the `MANIFEST.MF` file (you can also specify it at deployment time).

Configuring WebLogic Data Sources to Use Editions

The following list describes the different ways you can configure WebLogic data sources to use Oracle database editions.

- **Packaged Data Source Using a Single Edition**—The recommended way to configure the data source is to use a packaged data source descriptor that is stored in the application EAR or WAR file so that everything is self-contained. By doing so, you can use the same name for each data source and you do not need to change the application to use a variable name based on the edition. The data source URL in the descriptor should

reference the database service associated with the correct edition. If for some reason you are using a SID instead of a database service (no longer recommended), the alternative is to specify `SQL ALTER SESSION SET EDITION = name` in the Init SQL parameter in the data source descriptor. This SQL statement is executed for each newly created physical database connection in the data source pool. This approach assumes that a data source references only a single edition of the database and all connections use that edition.

Note the following restrictions when using a packaged data source.

- You cannot use a packaged data source with Logging Last Resource (LLR). You must use a system resource.
- You cannot use an application-scoped packaged data source with EmulateTwoPhaseCommit for the global-transactions-protocol with a versioned application. You must use a global-scoped data source.

  Therefore, if you need to use LoggingLastResource or EmulateTwoPhaseCommit, you cannot use this approach. See JDBC Application Module Limitations.

- **System Resource Data Source Using a Single Edition**—You can use a system resource as an alternative to a packaged data source. In this case, each data source must have a unique name and JNDI name. The application needs to be flexible enough to use that name at runtime. For example, you can pass in the data source JNDI name as a system property and the code that looks up the data source in JNDI will use that value.

  The disadvantage of using a single edition per data source, whether packaged or as a system resource, is that it requires more database connections. A single edition approach can work when the period during which the old and new editions are running is relatively short. For applications that are using a lot of data sources and/or connections, this is not a viable approach.

- **System Resource Data Source Using Multiple Editions**—An alternative is to have a data source that references multiple editions. The recommended configuration would still use a database service associated with a single edition. However, the connections will be re-associated with different editions during the lifetime of the connection.
- **Multiple Editions by Setting the Edition for Every Reservation**—It is possible for the application to set the database edition every time it gets a connection. There is some overhead associated with making this call each time (round trip to the database server and setting the session) and the application code needs to be modified everywhere that a connection is reserved. If you are using the JDBC Replay Driver, this initialization should be done in the `ConnectionInitializationCallback`. See Using a Connection Callback.

  It's important to optimize for the normal use case instead of optimizing for the (hopefully) short period during which the migration is done to a new edition. This approach doesn't optimize for the normal case where all connections are on the needed edition.

- **Multiple Editions using Connection Labeling**—You can also associate an edition with the connection and try to reserve a connection with the correct edition. The recommended way to tag a connection with a property is to use connection labeling. The application then needs to implement the pieces associated with connection labeling.
  - When a connection is reserved, it needs to determine the edition needed in the context.
  - A matching method is needed to determine if the property, in this case just the edition, matches.
  - A labeling initialization method is needed to make the connection match if it doesn't already match by using `SQL ALTER SESSION SET EDITION = name`.

  There is overhead associated with connection labeling, particularly when exclusively scanning the list of existing connections to find a mach. On the other hand, the normal use

case is that every connection matches the current edition so there is no need to look far to find a match. It is only during migration that there will be thrashing between editions and potentially longer searches to find a match (or to determine that there is no match).

Configure Oracle Parameters

WebLogic Server provides several attributes that provide improved data source performance when using Oracle drivers.

For detailed information, see Advanced Configurations for Oracle Drivers and Databases.

Configure ONS Client Parameters

ONS client configuration allows the data source to subscribe to and process Oracle FAN events.
When configuring the ONS node list, Oracle recommends not specifying a value and allowing auto-ONS to perform the ONS configuration. In some cases, however, it is necessary to explicitly configure the ONS configuration, for example if you need to specify an Oracle Wallet and password, or if you want to explicitly specify the ONS topology.

You can configure an ONS client using the following option:

- Configuring an ONS Client using WLST

Tuning Generic Data Source Connection Pools

You can improve application and system performance by ensuring a proper configuration of the connection pool attributes in JDBC data sources in your WebLogic Server domain.
For more information, see Tuning Data Source Connection Pools.

Generic Data Source Handling for Oracle RAC Outages

It is possible to use a Generic data source with Oracle RAC with some limitations. These limitations complicate transaction processing, monitoring, and graceful handling of RAC outages.

Note:

Oracle recommends using a Multi Data Source (MDS) or Active GridLink (AGL) data source instead of a Generic data source using driver-level failover. See Using Active GridLink Data Sources or Using Multi Data Sources with Oracle RAC.

The following limitations are due to Genetic data sources not being aware of the RAC instances associated with the connections in the pool:

- A Generic data source does not have the ability to disable a single instance in the pool that a MDS or AGL data source provides. If one of the RAC instances goes down (planned or unplanned), the data source tests all connections in the pool for the down instance, disabling them individually. In addition to more overhead and application delays, the pool sees multiple failures which cause the entire pool to be disabled. To prevent the pool from being disabled, set the value of `Count Of Test Failures Till Flush` to 0. See JDBCConnectionPoolParamsBean in *MBean Reference for Oracle WebLogic Server*.

- JTA or global transactions should not be used with this configuration. Because a Generic data source is not aware of the RAC instances, it cannot guarantee transaction affinity. This is a problem if the transaction spans multiple servers or if a failure occurs such that another connection is used to complete the transaction. Since the additional connections required to complete the transaction may not be within the same RAC instance, transaction processing may fail.
- It is not possible to monitor the connections based on the RAC instances.

Generic Data Source Handling of Driver-Level Failover

Several database drivers support a feature to define multiple database instances in the URL and failover from one database to the next. It is possible to use a Generic data source with driver-level failover with some limitations. These limitations complicate transaction processing, monitoring, and graceful handling of database instance outages.
The following limitations are due to WebLogic Server instances not being aware of the database instances associated with the connections in the pool:

- A Generic data source does not have the ability to disable a single instance in the pool that a Multi Data Source provides. If one of the database instances goes down (planned or unplanned), the data source tests all connections in the pool for the down instance, disabling them individually. In addition to more overhead and application delays, the pool sees multiple failures which cause the entire pool to be disabled. To prevent the pool from being disabled, set the value of `Count Of Test Failures Till Flush` to `0`.

  For more information, see JDBCConnectionPoolParamsBean in *MBean Reference for Oracle WebLogic Server*.
- JTA or global transactions should not be used with this configuration. Because WebLogic Server is not aware of the database instances, it cannot guarantee transaction affinity. This is a problem if the transaction spans multiple servers or if a failure occurs such that another connection is used to complete the transaction. Since the additional connections required to complete the transaction may not be within the same database instance, transaction processing may fail.
- It is not possible to monitor the connections based on the database instances.

4 JDBC Data Sources Types

Oracle WebLogic Server provides different types of JDBC data sources such as Generic data source, Multi Data Sources, and so on. You can configure database connectivity by configuring JDBC data sources and then targeting or deploying the JDBC resources to servers or clusters in your WebLogic domain.

- Using the Default Data Source
  Oracle provides a default data source required by a Jakarta EE 8-compliant runtime. This pre-configured data source can be used by an application to access the Derby Database installed with WebLogic Server.
- Using Generic Data Sources
- Using JDBC Multi Data Sources
- Using Active GridLink Data Sources
- Using Universal Connection Pool Data Sources
  A Universal Connection Pool (UCP) data source is provided as an option for users who wish to use Oracle Universal Connection Pooling to connect to Oracle Databases. UCP provides an alternative connection pooling technology to Oracle WebLogic Server connection pooling.

Using the Default Data Source

Oracle provides a default data source required by a Jakarta EE 8-compliant runtime. This pre-configured data source can be used by an application to access the Derby Database installed with WebLogic Server.

- What is Default Data Source
  Oracle provides a default data source required by a Jakarta EE 8-compliant runtime.
- Defining a Custom Default Data Source
  You can implement a custom default data source by defining a custom data source descriptor that is bound to `java:comp/DefaultDataSource` or overidding the default data source to point to another JNDI name.
- Compatibility Limitations When Using a Default Data Source
  Learn about the limitations when using a default data source.

What is Default Data Source

Oracle provides a default data source required by a Jakarta EE 8-compliant runtime.

It is accessible under the JNDI name:

```
java:comp/DefaultDataSource
```

which is equivalent to:

```
@Resource(lookup="java:comp/DefaultDataSource")
DataSource myDS;
```

You can explicitly bind a Data Source resource reference to the Default data source using the lookup element of the resource annotation or the lookup-name element of the `resource-ref` deployment descriptor element.

Note:

The Derby database is started by the `startWebLogic` command by default. For more information on starting and stopping a WebLogic Server instance, see Starting and Stopping Servers in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

Characteristics of a Default Data Source

A default data source has the following characteristics:

- Must be available for each component that is deployed.
- Only accessible for deployed components, not for data sources that are system resources or stand-alone deployments.
- Only visible in a console after it has been referenced.
- Appears as a deployment for each component, like other Jakarta EE deployments.
- Not configurable.
- Has the lifecycle of the associated application.

Configuring a Default Data Source

The following table provides the configuration settings that define the WebLogic Server default data source definition:

Table 4-1 Default Data Source Configuration

Attribute	Value
Name	`java:comp/DefaultDataSource`
Initial capacity	0
Min capacity	0
Max capacity	15
Classname	`org.apache.derby.jdbc.ClientDataSource`
Port	1527
Host	localhost
Database name	DefaultDataSource
User	none
Password	none
Transactional	false
MaxStatements	0

Table 4-1 (Cont.) Default Data Source Configuration

Attribute	Value
MaxIdleTimeout	not set

Defining a Custom Default Data Source

You can implement a custom default data source by defining a custom data source descriptor that is bound to `java:comp/DefaultDataSource` or overidding the default data source to point to another JNDI name.

See:

- Creating a Custom Default Data Source Descriptor
- Overriding the Default Data Source

After the component is deployed, if `java:comp/DefaultDataSource` is not available for the component, the WebLogic Server preconfigured default data source is available to the component. However, if you disabled the Derby database by setting `DERBY_FLAG=false)` before running `startWebLogic.sh` script, the WebLogic Server preconfigured default data source is not available.

Creating a Custom Default Data Source Descriptor

You can configure a data source descriptor that is bound to `java:comp/DefaultDataSource` replacing the preconfigured default data source. For example, the following provides an example of Jakarta EE 8 annotations in a EAR application:

```
@Stateless(mappedName="DSBean")
@DataSourceDefinition(name="java:comp/DefaultDataSource",
className="oracle.jdbc.OracleDriver",
portNumber=1521,
serverName="myServer",
databaseName="myDB",
user="a username",
password="a password",
transactional=false
)
public class DSBean implements DSInterface
. . .
```

Overriding the Default Data Source

You can override the preconfigured default data source provided by WebLogic Server by updating the JNDI name in the default data source attribute in the configuration of a server or server template to point to another existing data source.

See the following topics in Administering Oracle WebLogic Server with Fusion Middleware Control:

- Define general server configuration
- Configure server template general settings

Compatibility Limitations When Using a Default Data Source

Learn about the limitations when using a default data source.

In releases prior to Weblogic Server 12.2.1, WebLogic Server tries to satisfy unresolved data source `res-ref` references automatically by attempting to lookup the data source in JNDI using the name of the `res-ref`. This behavior is undefined prior to Jakarta EE 8. This WebLogic Server release uses the default data source as defined by Jakarta EE 8.

Using Generic Data Sources

Generic data sources provide database access and database connection management. Generic data sources and their connection pools provide connection management processes that help keep your system running efficiently.

- What is Generic Data Source
- Configuring Generic Data Source
  This topic describes the steps required to create and configure Generic data sources.

What is Generic Data Source

Generic 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.

Configuring Generic Data Source

This topic describes the steps required to create and configure Generic data sources.

Configure JDBC Data Source Properties

Data Source Names: You can use JDBCA data source name to identify the data source within the WebLogic domain. For system resource data sources, names must be unique among all other JDBC system resources. To avoid naming conflicts, data source names should also be unique among other configuration object names, such as servers, applications, clusters, and JMS queues, topics, and servers. For JDBC application modules packaged in an application, data source names must be unique among JDBC data sources with a similar scope.

The data source name cannot contain the following special characters: `@ # $`.

Data Source Scope: You can select the scope for the data source and set the scope to Global (at the domain level), or to any existing Resource Group or Resource Group Template.

JNDI Names: You can configure a data source so that it binds to the JNDI tree with a single or multiple names. See Using WebLogic JNDI in a Clustered Environment in *Developing JNDI Applications for Oracle WebLogic Server*.

Database Type: You can select the Database Management System (DBMS) of the database you want to connect. See Supported Configurations in *What's New in Oracle WebLogic Server*.

JDBC Driver: You must select a JDBC database driver that is preferred to create a database connection. You should verify, however, that the URL is as you want it before asking the console to test it. 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 WebLogic Remote Console are shipped (and/or are already in the `classpath`) with WebLogic Server. See Types of JDBC Drivers.

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`.

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. See Supported Configurations in *What's New in Oracle WebLogic Server*.

Configure Transaction Options

When you configure a JDBC data source using the WebLogic Remote Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver. XA, non-XA, and Global transaction options are supported by WebLogic JDBC data sources. See JDBC Data Source Transaction Options.

Configure Testing Options

You can set database connection testing options in a data source to make sure that the database connections remain healthy, which helps keep your applications running properly.

Connection Properties are used to configure the connection between the data source and the DBMS. Typical attributes are the database name, host name, port number, user name, and password.

Test Database Connection allows you to test a database connection before the data source configuration is finalized using a table name or SQL statement. If necessary, you can test additional configuration information using the Properties and System Properties attributes. See *Oracle WebLogic Remote Console Online Help*.

Configure Oracle Parameters

WebLogic Server provides several attributes that provide improved data source performance when using Oracle drivers. See Advanced Configurations for Oracle Drivers and Databases.

Target JDBC Data Sources

You can select one or more targets to which to deploy your new JDBC data source. If you don't select a target, the data source will be created but not deployed. You will need to deploy the data source at a later time before getting connections. See *Oracle WebLogic Remote Console Online Help* and Using JDBC Drivers with WebLogic Server.

Using JDBC Multi Data Sources

A Multi Data Source (MDS) is an abstraction around a group of Generic data sources that is bound to the JNDI tree or local application context just like Generic data sources are bound to the JNDI tree. You can configure a MDS to provide load balancing or failover processing at the time of connection requests, between the Generic data sources associated with the MDS.
For information about Generic data sources, see Using Generic Data Sources.

Applications lookup a MDS on the JNDI tree or in the local application context (`java:comp/env`) just as they do for generic data sources, and then request a database connection. The MDS determines which generic data source to use to satisfy the request depending on the algorithm selected in the MDS configuration: load balancing or failover.

Note:

Active GridLink and Multi Data Source are designed to work with Oracle RAC clusters. Oracle does not recommend using Generic data sources with Oracle RAC clusters. See Generic Data Source Handling for Oracle RAC Outages.

- What is Multi Data Source
- Configuring Multi Data Sources
- Multi Data Source Failover Enhancements
- Planned Database Maintenance with a Multi Data Source
  Learn how to handle planned maintenance, without service interruption, on the database server used by a Multi Data Source.
- Shutting Down the Data Source
  Shutting down the data source involves first suspending the data source and then releasing the associated resources including the connections.

What is Multi Data Source

Multi Data Source is used for failover or load balancing between nodes of a highly available database system such as Oracle Real Application Clusters (Oracle RAC). The Generic data source member list for a MDS source supports dynamic updates. This feature allows Oracle RAC environments to add and remove database nodes and corresponding Generic data sources without redeployment, grow and shrink RAC clusters in response to throughput, and shutdown Oracle RAC node for maintenance.

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.

Adding and removing database nodes is a manual operation performed by the database administrator.

- Adding a Database Node
  You can add a database node and corresponding Generic data sources without redeployment. This capability provides you the ability to start a node after maintenance or grow a cluster.
- Removing a Database Node
  You can remove a database node and corresponding Generic data sources without redeployment. This capability provides you the ability to shutdown a node for maintenance or shrink a cluster.

Adding a Database Node

You can add a database node and corresponding Generic 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 Generic data source.
3. Add the Generic data source back to the Multi Data Source.

Removing a Database Node

You can remove a database node and corresponding Generic 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 Generic data source from the Multi Data Source.
2. When all transactions have completed, suspend the Generic data source.
3. When all transactions have completed, shut down the Generic data source.
4. Shut down the database node.

Configuring Multi Data Sources

Perform the steps mentioned in this topic to create and configure Multi Data Source.

1. Create Generic data sources. See Using Generic Data Sources.
2. Create the Multi Data Source using either the WebLogic Remote Console or the WebLogic Scripting Tool. See *Oracle WebLogic Remote Console Online Help*.
3. Assign the Generic data sources to the Multi Data Source.

   For information about the configuration files created when configuring a Multi Data Source, see Understanding JDBC Resources in WebLogic Server and Creating a Multi Data Source Module.

> **Note:**
>
> In general, if a WebLogic Server data source setting of initial capacity is set to Zero, WebLogic Server makes no DBMS connections at startup. But to startup a Multi Data Source of LLR data sources, WebLogic Server makes a connection at startup to see if the DBMS is a RAC or not. For a generic LLR Multi Data Source, all the data sources need to be available, but if it is using RAC, only one node needs to be accessible for LLR processing.

- 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.
- Multi Data Source Fail-Over Limitations and Requirements
- Controlling Multi Data Source Failover with a Callback
  You can register a callback handler with WebLogic Server that controls when a MDS with the Failover algorithm fails over connection requests from one JDBC Generic data source in the MDS to the next Generic data source in the list.
- Deploying JDBC Multi Data Sources on Servers and Clusters

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
- Load Balancing

Failover

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

> **Note:**
>
> This algorithm requires that Test Reserved Connections (`TestConnectionsOnReserve`) on the Generic data source is enabled. If enabled, a connection in the first Generic data source is tested to verify if the Generic data source is healthy. If the connection fails the test, the Multi Data Source uses a connection from the next Generic data source listed in the Multi Data Source. See Connection Testing Options for a Data Source 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 Generic data source in the list. The MDS selects Generic data sources to use to satisfy connection requests using a round-robin scheme. When the MDS provides a connection, it selects a connection from the Generic data source listed just after the last Generic data source that was used to provide a connection. Multi Data Sources that use the Load Balancing algorithm also fail over to the next Generic data source in the list if a database connection test fails and the connection cannot be replaced, or if the Generic data source is suspended.

Multi Data Source Fail-Over Limitations and Requirements

WebLogic Server provides a failover algorithm for Multi Data Sources so that if a Generic data source fails (for example, if the database management system crashes), your system can continue to operate. However, there are certain limitations and requirements you must consider when configuring the Multi Data Source.

- Test Connections on Reserve to Enable Fail-Over
- No Fail-Over for In-Use Connections

Test Connections on Reserve to Enable Fail-Over

Generic data sources rely on the Test Reserved Connections (`TestConnectionsOnReserve`) feature on the Generic data source to know when database connectivity is lost. Testing reserved connections must be enabled for the Generic 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 Generic data source in the Multi Data Source. After a test failure, the Generic data source attempts to recreate the connection. If that attempt fails, the Multi Data Source fails over to the next Generic 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 the application code close the failed connection, and the transaction must be restarted from the beginning with a new connection.

Controlling Multi Data Source Failover with a Callback

You can register a callback handler with WebLogic Server that controls when a MDS with the Failover algorithm fails over connection requests from one JDBC Generic data source in the MDS to the next Generic 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 using the **Failover Callback Handler** attribute of the MDS and are registered per MDS. You must register the callback handler for each MDS to which you want the callback handler to apply. And you can register different callback handlers for each MDS in your domain.

- Callback Handler Requirements
- Callback Handler Configuration
- How It Works—Failover

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 Generic data source in the list or when a previously disabled Generic 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 weblogic.jdbc.extensions.ConnectionPoolFailoverCallback interface.

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 Generic 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 WebLogic Remote Console (see *Oracle*

WebLogic Remote Console Online Help) 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) Generic data sources to see if they are now available. See Automatic Re-enablement on Recovery of a Failed Generic Data Source within a Multi Data Source for more details.

How It Works—Failover

WebLogic Server attempts to failover connection requests to the next Generic data source in the list when the current Generic data source fails a connection test or, if you enabled `FailoverRequestIfBusy`, when all connections in the current Generic 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 Generic data source in the list. If a connection from that Generic data source fails a connection test, WebLogic Server marks the Generic 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 Generic data source currently being used to supply database connections. This is the "failover from" Generic data source.
- `nextPool`—The name of next available Generic data source listed in the Multi Data Source. For failover, this is the "failover to" Generic data source.
- `opcode`—A code that indicates the reason for the call:
  - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current Generic data source is dead and has disabled it.
  - `OPCODE_CURR_POOL_BUSY`—All database connections in the Generic data source are in use. (Requires `FailoverIfBusy=true` in the Multi Data Source configuration. See Enabling Failover for Busy Generic Data Sources in a Multi Data Source.)

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 Generic data source in the list.
- `RETRY_CURRENT`—Retry the connection request with the current Generic 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 Generic data sources fails, WebLogic Server calls the callback handler again, as in the previous failover, in an attempt to failover to the next available Generic 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 Generic data source.

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

Deploying JDBC Multi Data Sources on Servers and Clusters

All Generic 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 Generic 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.

Multi Data Source Failover Enhancements

Learn how to improve failover processing for Multi Data Sources.

- Connection Request Routing Enhancements When a Generic Data Source Fails
- Automatic Re-enablement on Recovery of a Failed Generic Data Source within a Multi Data Source
- Enabling Failover for Busy Generic Data Sources in a Multi Data Source
- Controlling Multi Data Source Failback with a Callback

Connection Request Routing Enhancements When a Generic Data Source Fails

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

This feature requires that Generic data source testing options are configured for allGeneric data sources in a Multi Data Source, specifically Test Table Name and Test Reserved Connections. See Connection Testing Options for a Data Source.

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

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

After a Generic data source is automatically disabled because a connection failed a connection test, the Multi Data Source periodically tests a connection from the disabled Generic data source to determine when the Generic data source (or underlying database) is available again. When the Generic data source becomes available, the Multi Data Source automatically re-enables the Generic data source and resumes routing connection requests to the Generic data source, depending on the Multi Data Source algorithm and the position of the Generic data source in the list of included Generic 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 Generic data sources every 120 seconds.

WebLogic Server does not test and automatically re-enable Generic data sources that you manually disable. It only tests Generic 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 Generic data source. See Controlling Multi Data Source Failback with a Callback for more details.

Enabling Failover for Busy Generic 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 Generic 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 Generic data source are in use, you can enable the **Failover Request if Busy** option in the WebLogic Remote Console. (Also available as the `FailoverRequestIfBusy` attribute in the JDBCDataSourceParamsBean). If enabled (set to `true`), when all connections in the current Generic data source are in use, application requests for connections will be routed to the next available Generic 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 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 Generic data source that was automatically disabled. You can use the callback to control if or when the disabled Generic data source is re-enabled so that you can make any other system preparations before the Generic 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:

- How It Works—Failback

How It Works—Failback

WebLogic Server periodically checks the status of Generic data sources in a Multi Data Source that were automatically disabled. (See Automatic Re-enablement on Recovery of a Failed

Generic Data Source within a Multi Data Source.) If a disabled Generic 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 Generic 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 Generic data source named in `currPool` is now available.

Failback, or automatically re-enabling a disabled Generic 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 Generic data source. WebLogic Server resumes routing connection requests to the Generic data source, depending on the Multi Data Source algorithm and the position of the Generic data source in the list of included Generic data sources.
- `DONOT_FAILOVER`—Do not re-enable the `currPool` Generic data source. Continue to serve connection requests from the Generic data sources 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 Generic 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 Generic 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 Generic data source in the list of Generic 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 Generic 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 Generic data source. Future connection requests will be served by `DS1` because `DS1` is the first Generic 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 Generic data source, connection requests will continue to be served by `DS1` because `DS1` is listed before `DS2` in the list of Generic data sources.

Planned Database Maintenance with a Multi Data Source

Learn how to handle planned maintenance, without service interruption, on the database server used by a Multi Data Source.

To avoid service interruption, multiple database instances must be available so that the database can be updated in a rolling fashion. Oracle RAC cluster and Oracle GoldenGate, or a combination of these products, can be used to help accomplish this goal. (Note that Oracle DataGuard cannot be used for planned maintenance without service interruption). Each database instance is configured as a Generic data source member of the Multi Data Source. This approach assumes that the application is returning connections to the pool on a regular basis.

Process Overview

The following steps provide a high-level overview of the planned maintenance process:

1. On mid-tier systems—Shutdown all member data sources associated with the Oracle RAC instance that will be shut down for maintenance. It is important that you do not shut down all data sources in each Multi Data Source list so that connections can be reserved for the other member(s). Wait for data source shutdown to complete. See:
   - Shutting Down the Data Source
   - JDBCDataSourceRuntimeMBean shutdown operation in *MBean Reference for Oracle WebLogic Server*
2. If required, you may want to reduce the remaining connections on the database side that are not associated with the WebLogic data source. For the Oracle database server, this might include stopping (or relocating) the application services at the instances that will be shut down for maintenance, stopping the listener, and/or issuing a transactional disconnect for the services on the database instance.
3. Shut down the database instance using your preferred tools.
4. Perform the planned maintenance
5. Restart the database instance using your preferred tools.
6. Startup the services when the database instances are ready for application use.
7. On mid-tier systems—Start the member data sources. See JDBCDataSourceRuntimeMBean start operation in *MBean Reference for Oracle WebLogic Server*.

Shutting Down the Data Source

Shutting down the data source involves first suspending the data source and then releasing the associated resources including the connections.

When a member data source in a Multi Data Source is marked as suspended, the Multi Data Source will not try to get connections from the suspended pool. Instead, to reserve connections, it will go to the next member data source. It is important that you do not shut down all member data sources in a Multi Data Source list at the same time. If all members are shut down or fail, then access to the Multi Data Source fails and the application will see failures.

When you gracefully suspend a data source, which is the first step of the shut down process, the following occurs:

- The data source is immediately marked as suspended at the beginning of the operation and no further connections are created on the data source.
- Idle (not reserved) connections are marked closed
- After a timeout period for the suspend operation, all remaining connections in the pool are marked as suspended and the following exception is thrown for any operations on the connection, indicating that the data source is suspended:

  `java.sql.SQLRecoverableException: Connection has been administratively disabled. Try later.`

- All the remaining connections are then closed. We won't know until the data source is resumed if they are good or not. In this case, we know that the database will be shut down and the connections in the pool will not be good if the data source is resumed. Instead, we are doing a data source shutdown which will close all of the disabled connections.

  The shutdown operation can be done synchronously or asynchronously. If you do a synchronous shutdown, the default timeout period is 60 seconds. You can change the value of this timeout period by configuring or dynamically setting `Inactive Connection Timeout Seconds` to a non-zero value. There is no upper limit on the inactive timeout period. Note that the processing actually checks for in-use (reserved) resources every tenth of a second so if the timeout value is set to 2 hours and all reserved resources are released a second later, the shut down will complete a second later. If you do an asynchronous operation, the timeout period is specified on the method itself. If set to 0, the default is used. The default is to use Inactive Connection Timeout Seconds if set or 60 seconds. If you want a minimal timeout, set the value to 1. If you want no timeout, set it to a large value (not recommended).

This shutdown operation runs synchronously; there is no asynchronous version of the MBean operation available.

You can also use this for Multi Data Sources configured with either Load-Balancing or Failover.

Example 4-1 WLST Example

The following WLST example script demonstrates how to edit the configuration to increase the suspend timeout period and then use the runtime MBean to shutdown a data source. This script must be integrated into the existing framework for all WebLogic Server servers and data sources.

```
import sys, socket, os
hostname = socket.gethostname()
datasource='ds'
svr='myserver'
connect("weblogic","password","t3://"+hostname+":7001")
# Shutdown the data source serverRuntime()
cd('/JDBCServiceRuntime/' + svr + '/JDBCDataSourceRuntimeMBeans/' +
datasource )
task = cmo.shutdown(10000)
while (task.isRunning ()):
  print 'SHUTTING DOWN';
  java.lang.Thread.sleep(2000);
  print 'Datasource task is in status' + task.getStatus();
exit()
$ java weblogic.WLST myscript2.py
Intializing Weblogic Scripting Tool (WLST)...
Welcome to WebLogic Server Administration Scripting Shell
....
Location changed to serverRuntime tree.
```

```
This is a read-only tree with ServerRuntimeMBean as the root. For more help,
use help('serverRuntime').
SHUTTING DOWN
Datasource task is in status
SUCCESS
Datasource task is in status
SUCCESS
Exiting WebLogic Scripting Tool.
```

Important Considerations

The following list describes issues you should be aware of when performing planned maintenance:

- If the Multi Data Source is using a database service, you cannot stop or relocate the database service before suspending or shutting down the Multi Data Source. If you do, the Multi Data Source may attempt to create a connection to the now missing service and it will react as though the database is down and kill all connections, preventing a graceful shutdown. Because suspending a Multi Data Source ensures that no new connections are created at the associated instance, it is not necessary to stop the service. (Note that the Multi Data Source only creates connections on this instance. It will never create connections on another instance even if it is relocated). Also, suspending a Multi Data Source ceases operations on all connections, therefore no further progress occurs on any sessions (the transactions will not complete) that remain in the Multi Data Source pool.
- You may encounter an issue related to XA affinity that is enforced by the Multi Data Source algorithms. When an XA branch is created on an Oracle RAC instance, all additional branches are created on the same instance. While Oracle RAC supports XA across instances, there are some significant limitations that applications run into before the prepare phase, and the Multi Data Source enforces that all operations are on the same instance. As soon as the graceful suspend operation starts, the member data source is marked as suspended and no further connections are allocated there. If an application using global transactions tries to start another branch on the suspending data source, it will fail to get a connection and the transaction fails. In the case of an XA transaction spanning multiple WebLogic servers, the suspend is not graceful. This issue does not apply to Emulate Two-Phase Commit or one-phase commit, which use a single connection for all work, and Logging Last Resource (LLR).
- If for some reason you must separate suspending the data source, at which point all connections are disabled, from releasing the resources, you can perform a suspend followed by `forceShutdown`. You must use a forced shutdown to avoid going through the waiting period a second time. Oracle does not recommend using this process.
- To get a graceful shutdown of the data source when shutting down the database, the data source must be involved. This process of shutting down the data source followed by shutdown of the database requires coordination between the mid-tier and the database server processing. Processing is simplified by using Active GridLink instead of Multi Data Source. See Using Active GridLink Data Sources.
- When using the Oracle database, Oracle recommends that an application service be configured for each database so that it can be configured for high availability. By using an application service, you can start up the database on its own without the data source starting to use it. Once the application service is explicitly started, the administrator can make the database available to the data source.

Using Active GridLink Data Sources

An Active GridLink (AGL) data source provides connectivity between WebLogic Server and an Oracle database. Oracle database offers both on-premises and cloud database services with cluster capabilities of Oracle Grid Infrastructure and Oracle Clusterware.
For more information, see Supported Oracle On-Premises and Cloud Database Services and Understanding the ActiveGridlink Attribute.

Using an AGL data source involves creating the AGL data source, configuring the connection pool and Oracle database parameters, tuning, monitoring, and so on. The following sections explain in detail these concepts:

- What is Active GridLink Data Source
- Configuring Active GridLink Data Source
  Use the WebLogic Remote Console or WLST to configure Active GridLink Data Source in a WebLogic domain.
- Configuring Runtime Load Balancing using SDP
  To configure load balancing across SDP connections, you must edit the `TNSNAMES.ORA` file on all nodes and add an SDP end-point to the `LISTENER_IBLOCAL` entry.
- Configuring Active GridLink Connection Pool Features
- Tuning Active GridLink Data Source Connection Pools
  By properly configuring the connection pool attributes in JDBC data sources in your WebLogic Server domain, you can improve application and system performance.
- Monitoring Active GridLink JDBC Resources
  Learn about monitoring and debugging Active GridLink data sources.
- Using Active GridLink Data Sources without FAN Notification
- Best Practices for Active GridLink Data Sources
  Learn about the best practices for using Active GridLink data sources by understanding the catch and handle exceptions and how connections are created when using an Active GridLink data source.
- Comparing Active GridLink and Multi Data Sources
  There are several benefits to using Active GridLink data sources over Multi Data Sources when using Oracle RAC clusters.
- Migrating from Multi Data Source to Active GridLink
  You can migrate to Multi Data Source from Active GridLink data sources using simple manual process.
- Managing Database Downtime with Active GridLink Data Sources
  Learn several ways to handle database downtime with Active GridLink data sources in an Oracle RAC database environment.
- Gradual Draining
  During planned database maintenance, gradually close the database connections instead of closing all of the connections immediately. This strategy prevents uneven performance by the application.

What is Active GridLink Data Source

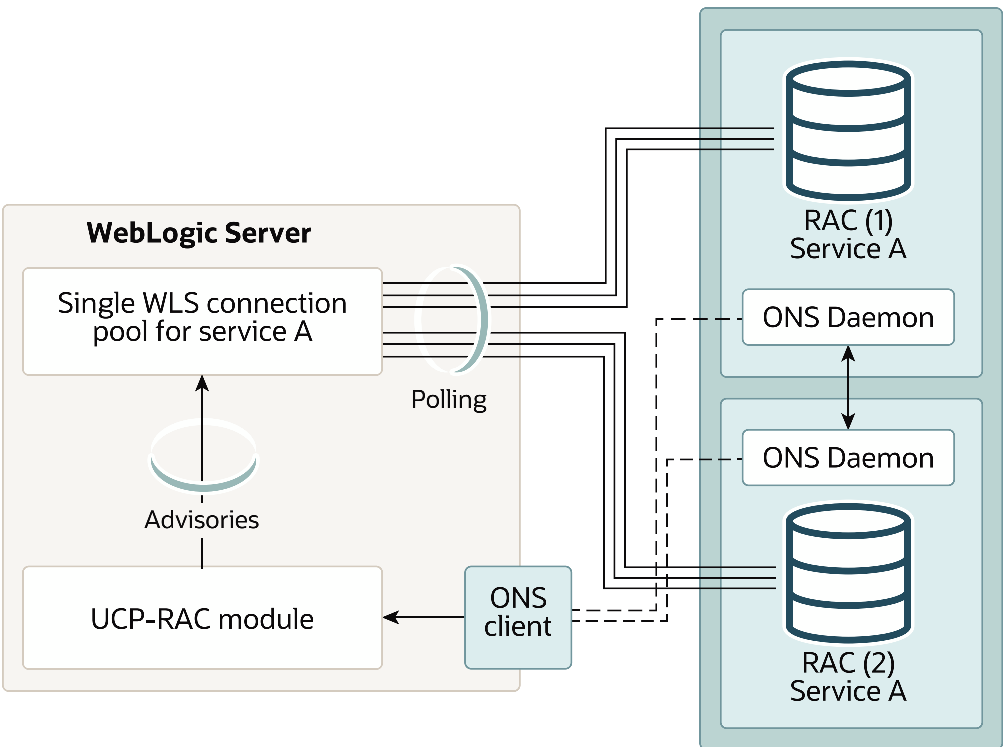
An Active GridLink Data Source (AGL) data source provides connectivity between WebLogic Server and an Oracle database service, which may include one or more Oracle RAC clusters.

An Oracle database service represents a workload with common attributes that enables system administrators to manage the workload as a single entity.
You scale the number of AGL data sources as the number of services increases in the data base, independent of the number of nodes in the Oracle RAC cluster(s). Examples of High Availability support for multiple clusters include Data Guard, GoldenGate, and Global Database Service.

Note:

Active GridLink and Multi Data Source are designed to work with Oracle RAC clusters. Oracle does not recommend using Generic data sources with Oracle RAC clusters. See Comparing AGL and Multi Data Sources.

Figure 4-1 Active GridLink Data Source Connectivity



An Active GridLink data source includes the features of Generic data sources plus the following support for Oracle RAC:

- Fast Connection Failover
- Runtime Connection Load Balancing
- GridLink Affinity
- SCAN Addresses
- Secure Communication using Oracle Wallet with ONS Listener
- Support for Active Data Guard

- Supported Oracle On-Premises and Cloud Database Services
- Using Socket Direct Protocol

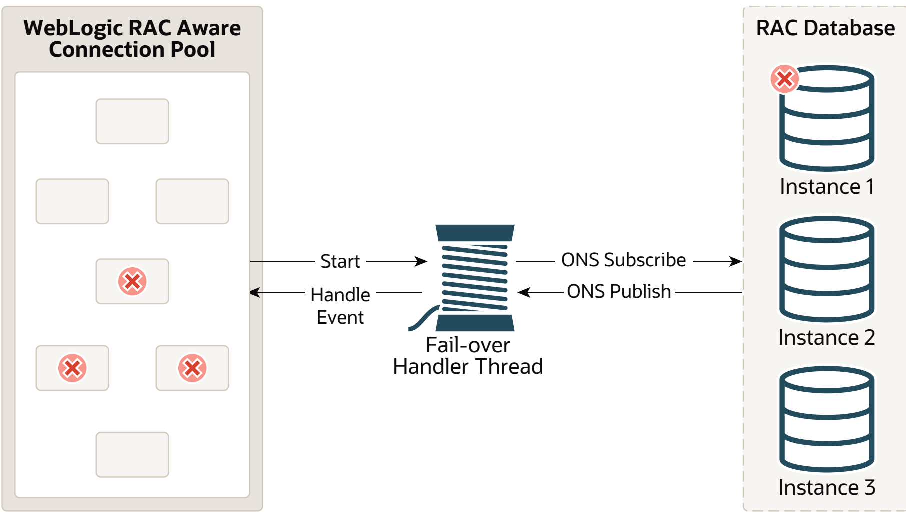
Fast Connection Failover

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

WebLogic Server supports Fast Connection Failover. See About Fast Connection Failover in *Universal Connection Pool for JDBC Developer's Guide*.

An AGL data source uses Fast Connection Failover and responds to Oracle RAC events using Oracle Notification Service (ONS). This ensures that the connection pool in the AGL data source contains valid connections (including reserved connections) without the need to poll and test connections. It also ensures that connections are created on new nodes as they become available.

Figure 4-2 Fast Connection Failover



An AGL data source uses Fast Connection Failover to:

- Provide rapid failure detection.
- Abort and remove invalid connections from the connection pool.
- Perform graceful shutdown for planned and unplanned Oracle RAC node outages. See Planned Outage Procedures and Unplanned Outages.
- Adapt to changes in topology, such as adding or removing a node.
- Distribute runtime work requests to all active Oracle RAC instances, including those rejoining a cluster.

> **Note:**
>
> AGL data sources do not support the deprecated `FastConnectionFailoverEnabled` connection property. An attempt to create an XA connection with this property enabled results in a `java.sql.SQLException: Can not use getXAConnection() when connection caching is enabled` exception because the driver implementation of Fast Connection Failover for this property does not support XA connections.

- JDBC Driver Configuration for use with Oracle Fast Connection Failover
  To enable Fast Connection Failover on a data source, you need to set specific values for the Driver Class Name and ONS configuration string properties.

JDBC Driver Configuration for use with Oracle Fast Connection Failover

To enable Fast Connection Failover on a data source, you need to set specific values for the Driver Class Name and ONS configuration string properties.

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 Oracle RAC nodes to Oracle FAN/ONS events. For example:
  `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.

Runtime Connection Load Balancing

AGL data sources provide load balancing. AGL data sources use runtime connection load balancing (RCLB) to distribute connections to Oracle RAC instances based on Oracle FAN events issued by the database. This simplifies data source configuration and improves performance as the database drives load balancing of connections through the AGL data source, independent of the database topology.

Runtime Connection Load Balancing allows WebLogic Server to:

- Adjust the distribution of work based on back end node capacities such as CPU, availability, and response time.
- React to changes in Oracle RAC topology.
- Manage pooled connections for high performance and scalability.

Figure 4-3 Runtime Connection Load Balancing

If FAN is not enabled, AGL data sources use a round-robin load balancing algorithm to allocate connections to Oracle RAC nodes.

Note:

Connections may be shut down periodically on AGL data sources. If the connections allocated to various RAC instances do not correspond to the Runtime Load Balancing percentages in the FAN load-balancing advisories, connections to overweight instances are destroyed and new connections opened. This process occurs every 30 seconds by default.

You can tune this behavior using the `weblogic.jdbc.gravitationShrinkFrequencySeconds` system property which specifies the amount of time, in seconds, the system waits before rebalancing connections. A value of 0 disables the rebalancing process.

GridLink Affinity

WebLogic Server GridLink affinity policies are designed to improve application performance by maximizing RAC cluster utilization.

- Session Affinity Policy
- XA Affinity Policy

Session Affinity Policy

Web applications have better performance when repeated operations against the same set of records are processed by the same RAC instance. Business applications such as online shopping and online banking are typical examples of this pattern.

An AGL data source uses the Session Affinity policy to ensure all the data base operations for a web session, including transactions, are directed to the same Oracle RAC instance of a RAC cluster.

> **Note:**
>
> The context is stored in the HTTP session. It is up to the application how windows (within a browser or across browsers) are mapped to HTTP sessions.

If an AGL data source with a session affinity policy is accessed outside the context of a web session, the affinity policy changes to the XA affinity policy. See XA Affinity Policy.

Figure 4-4 Session Affinity

An AGL data source monitors RAC load balancing advisories (LBAs) using the AffEnabled attribute to determine if RAC affinity is enabled for a RAC cluster. The first connection request is load balanced using Runtime Connection Load-Balancing (RCLB) and is assigned an Affinity context. All subsequent connection requests are routed to the same Oracle RAC instance using the Affinity context of the first connection until the session ends or the transaction completes. Affinity is based on the database name, service name, and instance name. Although the Session Affinity policy for an AGL data source is always enabled by default, a Web session is active for Session Affinity if:

- Oracle RAC is enabled, active, and the service has enabled RCLB. RCLB is enabled for a service if the service `GOAL` (NOT `CLB_GOAL`) is set to either `SERVICE_TIME` or `THROUGHPUT`.
- The database determines there is sufficient performance improvement in the cluster wait time and the Affinity flag in the payload in the information from ONS is set to `TRUE`.

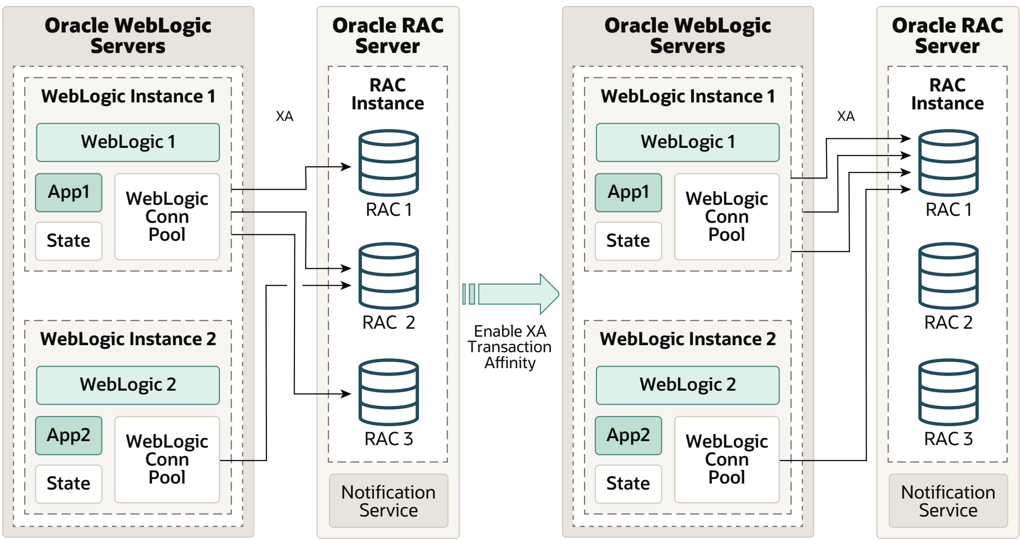
If the database determines it is not advantageous to implement session affinity, such as a high database availability condition, the database load balancing algorithm reverts to its default work allocation policy and the Affinity flag in the payload is set to `FALSE`.

XA Affinity Policy

XA Affinity for global transactions ensures all the data base operations for a global transaction performed on an Oracle RAC cluster are directed to the same Oracle RAC instance. There are limitations to consider:

- XA transaction can't span instances.
- Strict affinity is enforced for connections within an XA transaction. If a connection cannot be created on the correct instance, an exception is thrown.

Figure 4-5   XA Affinity



SCAN Addresses

There are two options to load balance connections across nodes:

- Use a single Oracle Single Client Access Name (SCAN) address

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=scanname)
  (PORT=scanport))(CONNECT_DATA=(SERVICE_NAME=myservice)))
  ```

- Use multiple non-SCAN addresses with LOAD_BALANCE=on

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=ON)
  (ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)
  (HOST=host2)(PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=myservice)))
  ```

Using a SCAN address is recommended over using multiple non-SCAN addresses. However, a SCAN address can only be used if your database is configured to use it. Contact your network administrator for appropriately configured SCAN URLs for your environment.

Note:

When using Oracle RAC 11.2 and higher, consider the following:

- If the Oracle RAC listener is set to `SCAN`, the AGL data source configuration can only use a SCAN address.
- If the Oracle RAC listener is set to `List of Node VIPs`, the AGL data source configuration can only use a list of VIP addresses.
- If the Oracle RAC listener is set to `Mix of SCAN and List of Node VIPs`, the AGL data source configuration can use both SCAN and VIP addresses.

See:

- Overview of Automatic Workload Management with Dynamic Database Services in *Real Application Clusters Administration and Deployment Guide*.
- Oracle Single Client Access Name (SCAN) White Paper at `http://www.oracle.com/technetwork/database/clustering/overview/scan-129069.pdf`

Secure Communication using Oracle Wallet with ONS Listener

This feature allows you to configure secure communication with the ONS listener using Oracle Wallet. See Secure ONS Client Communication.

Support for Active Data Guard

Active GridLink data source also works with Oracle Active Data Guard. Oracle Clusterware must be installed and active on the primary and standby sites for both single instance (using Oracle Restart) and Oracle RAC databases. Oracle Data Guard broker coordinates with Oracle Clusterware to properly fail over role-based services to a new primary database after a Data Guard failover has occurred. Cluster Ready Services (CRS) posts FAN events when the role change occurs.

Supported Oracle On-Premises and Cloud Database Services

Oracle database offers both on-premises and cloud database services that use the Fast Application Notification (FAN) feature provided with the cluster capabilities of Oracle Grid Infrastructure and Oracle Clusterware.

Oracle database on-premises services that use the FAN feature include the following products and features:

- Oracle Real Application Clusters (RAC). See, Using WebLogic Server with Oracle RAC.
- Oracle Real Application Clusters (RAC) One Node. See Overview of Oracle Real Application Clusters One Node and Administering Oracle RAC One Node in *Real Application Clusters Administration and Deployment Guide*.
- Oracle Data Guard (with Broker). See Oracle Data Guard Broker Concepts in *Oracle® Data Guard Broker Guide*.
- Oracle Standard Edition High Availability. See About Standard Edition High Availability and Installing Standard Edition High Availability in *Database Installation Guide*.
- Oracle Database Global Data Services. See Global Data Services.

Oracle Database related cloud services that use the FAN feature includes the following products:

- Oracle Autonomous Transaction Processing Dedicated (ATP-D). See About Dedicated Autonomous Database and Access Autonomous Database in the Oracle Cloud Infrastructure Console in *Oracle Autonomous Database on Dedicated Exadata Infrastructure Guide*.
- Oracle Autonomous Database Dedicated (ADB-D). See About Dedicated Autonomous Database in *Oracle Autonomous Database on Dedicated Exadata Infrastructure Guide*.
- Oracle Exadata Cloud@Customer. See About Oracle Exadata Cloud at Customer in *Exadata Database Service on Cloud@Customer Administrator's Guide*.
- Oracle Exadata Cloud Service. See About Exadata Cloud Service Instances in *Administering Oracle Database Exadata Cloud Service Guide*.
- Oracle Database Cloud Service. See About Oracle Database Cloud Services in *Administering Oracle Database Classic Cloud Service Guide*.

Using Socket Direct Protocol

To use the Socket Direct Protocol (SDP), your database network must be configured to use Infiniband. SDP does not support SCAN addresses.

See Configuring SDP Support for InfiniBand Connections in the *Oracle Database Net Services Administrator's Guide*.

Configuring Active GridLink Data Source

Use the WebLogic Remote Console or WLST to configure Active GridLink Data Source in a WebLogic domain.

See:

- *Oracle WebLogic Remote Console Online Help*
- The sample WLST script `EXAMPLES_HOME\wl_server\examples\src\examples\wlst\online\jdbc_data_source_creation.py`, where `EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. This example creates a Generic data source. See WLST Online Sample Scripts in *Understanding the WebLogic Scripting Tool*.

You must perform the following basic steps to create a data source using the WebLogic Remote Console:

- Configure JDBC Data Source Properties
- Configure Transaction Options
- Configure Connection Properties
- Test Connections
- Configure ONS Client
- Target the Data Source
- Configuring Oracle Parameters
- Configuring an ONS Client Using WLST

Configure JDBC Data Source Properties

JDBC Data Source Properties include options that determine the identity of the data source and the way the data is handled on a database connection.

- **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. To avoid naming conflicts, data source names should also be unique among other configuration object names, such as servers, applications, 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 that are similarly scoped. The data source name cannot contain the following special characters: `@ # $`.
- **Data Source Scope**: You can select the scope for the data source and set the scope to Global (at the domain level), or to any existing Resource Group or Resource Group Template.
- **JNDI Names**: You can configure a data source so that it binds to the JNDI tree with a single or 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. See *Developing JNDI Applications for Oracle WebLogic Server*.
- **Driver:** Select the replay driver for JDBC Replay Driver, or the XA or non-XA Thin driver.

> **Note:**
>
> The JDBC Replay Driver does not currently support XA transactions.

Configure Transaction Options

When you configure a JDBC data source using the WebLogic Remote Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver. WebLogic JDBC data sources support global transaction integration for XA drivers and non-XA drivers with the options Logging Last Resource, Emulate Two-Phase Commit, One-Phase Commit and no integration.

For more information on configuring transaction support for a data source, see JDBC Data Source Transaction Options.

Configure Connection Properties

Connection Properties are used to configure the connection between the data source and the DBMS. Typical attributes are the service name, database name, host name, port number, user name, and password.

Note:

Using service names:

- When a Database Domain is used, service names must be suffixed with the domain name. For example, if the database name is `db.country.myCorp.com`, the service name `myservice` would need to be entered as `myservice.db.country.myCorp.com`.

The console allows you to enter connection properties in one of the following ways:

- Enter Connection Properties
- Enter a Complete URL
- Supported Active GridLink Data Source URL Formats

Enter Connection Properties

On the **GridLink data source connection Properties Options** page, select **Enter individual listener information** and click **Next**. Enter the connection properties. For example:

- Enter **myService** in `Service Name`.
- Enter **left:1234**, **center:1234**, **right:1234** in the `Host and Port:`. Separate the host and port of each listener with colon.
- Enter **myDataBase** in `Database User Name`.
- Enter **myPassword1** in `Password`.
- If required, set **Protocol** to `SDP`.

The console automatically generates the complete JDBC URL. For example:

```
jdbc:oracle:thin:@( DESCRIPTION = (ADDRESS_LIST = (LOAD_BALANCE=on) (FAILOVER=ON)
(ADDRESS=(PROTOCOL=TCP)(HOST=left)(PORT=1521)) (ADDRESS=(PROTOCOL=TCP)
(HOST=center)(PORT=1521)) (ADDRESS=(PROTOCOL=TCP)(HOST=right)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME=myService)))
```

Enter a Complete URL

On the **GridLink data source connection Properties Options** page, select **Enter complete JDBC URL** and click **Next**. Enter the connection properties. For example:

- In **Complete JDBC URL**, enter the JDBC URL. For example:

  ```
  jdbc:oracle:thin:@( DESCRIPTION = (ADDRESS_LIST = (LOAD_BALANCE=on)
  (FAILOVER=ON) (ADDRESS=(PROTOCOL=TCP)(HOST=left)(PORT=1521))
  (ADDRESS=(PROTOCOL=TCP)(HOST=center)(PORT=1521)) (ADDRESS=(PROTOCOL=TCP)
  (HOST=right)(PORT=1521))) (CONNECT_DATA=(SERVICE_NAME=myService)))
  ```

  You can also use a SCAN address. For example:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
  (HOST=MyScanAddr-scn.myCompany.com)(PORT=1234)))
  (CONNECT_DATA=(SERVICE_NAME=myService)))
  ```

- Enter **myDataBase** in `Database User Name`.
- Enter **myPassword1** in `Password`.

- If required, set **Protocol** to `SDP`.

Supported Active GridLink Data Source URL Formats

AGL data sources only support long format JDBC URLs. The supported long format pattern is:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
(HOST=[SCAN_VIP])(PORT=[SCAN_PORT])))
(CONNECT_DATA=(SERVICE_NAME=[SERVICE_NAME])))
```

Easy Connect (short) format URLs are not supported for AGL data sources. The following is an example of a Easy Connect URL pattern that is not supported for use with AGL data sources:

```
jdbc:oracle:thin:[SCAN_VIP]:[SCAN_PORT]/[SERVICE_NAME]
```

Recommendations for AGL Data Source URLs

The following section provides general recommendations when creating AGL data source URLs.

- Use a single `DESCRIPTION`. Avoid a `DESCRIPTION_LIST` to avoid connection delays.
- Use one `ADDRESS_LIST` for each RAC cluster or DataGuard database
- Enter `RETRY_COUNT, RETRY_DELAY, CONNECT_TIMEOUT` at the `DESCRIPTION` level so that all `ADDRESS_LIST` entries use the same value.
- `RETRY_DELAY` specifies the delay, in seconds, between the connection retries. This attribute is new in the Oracle 12.1.0.2 release.
- `RETRY_COUNT` is used to specify the number of times an `ADDRESS` list is traversed before the connection attempt is terminated. The default value is 0. When using `SCAN` listeners with `FAILOVER=on`, setting `RETRY_COUNT` to a value of 2 means that if you had 3 `SCAN IP` addresses, each would be traversed three times each, resulting in a total of nine connect attempts (3 * 3)
- Specify `LOAD_BALANCE=on` for each address list to balance the `SCAN` addresses.
- The service name should be a configured application service, not a PDB or administration service.
- `CONNECT_TIMEOUT` is used to specify the overall time used to complete the Oracle Net connect. Set `CONNECT_TIMEOUT=90` or higher to prevent logon storms. For JDBC driver 12.1.0.2 and earlier, `CONNECT_TIMEOUT` is also used for the TCP/IP connection timeout for each address in the URL. When considering TCP/IP connections, a shorter `CONNECT_TIMEOUT` is preferred though secondary to overall timeout requirements.
- Do not set the `oracle.net.CONNECT_TIMEOUT` driver property on the data source because it is overridden by the URL property.

Test Connections

Test Database Connection allows you to test a database connection before the data source configuration is finalized using a table name or SQL statement. If necessary, you can test additional configuration information using the `Properties` and `System Properties` attributes.

Configure ONS Client

ONS client configuration allows the data source to subscribe to and process Oracle FAN events. When configuring the ONS node list, Oracle recommends not specifying a value and

allowing auto-ONS to perform the ONS configuration. In some cases, however, it is necessary to explicitly configure the ONS configuration, for example if you need to specify an Oracle Wallet and password, or if you want to explicitly specify the ONS topology.

You can also configure an ONS client using WLST. For an example, see Configuring an ONS Client Using WLST.

Other Considerations

In general, if a WebLogic Server data source setting of initial capacity is set to 0, WebLogic Server makes no DBMS connections at startup. For Active GridLink data sources with Auto-ONS, WebLogic Server needs to connect to the DBMS once at startup to get the ONS information.

- Enabling FAN Events
- Configure ONS Host and Port
- Secure ONS Client Communication
- Test ONS Client Configuration

Enabling FAN Events

To ensure that the data source is configured to subscribe to and process Oracle Fast Application Notification (FAN) events, select `Fan Enabled`.

Configure ONS Host and Port

There are two methods that you can use to configure the `OnsNodeList` value: a single node list or a property node list. You can use one or the other, but not both. If the WebLogic Server `OnsNodeList` contains an equals sign (=), it is assumed to be a property node list.

For both types of node lists you can use a Single Client Access Name (SCAN) address instead of a host name, and to access FAN notifications. For more information about SCAN addresses, see Scan Addresses.

To configure the `OnsNodeList` value using a:

- **Single node list**—Specify a comma separated list of ONS daemon listen addresses and ports for receiving ONS-based FAN events. For example, `rac1:6200,rac2:6200`. You can enter a single node list in the ONS host and port field in the Remote Console when creating an AGL Data Source.
- **Property node list**—Specify a string composed of multiple records, with each record consisting of a key=value pair and terminated by a new line ('\n') character. For example, nodes.`1=rac1:6200,rac2:6200`. You cannot enter a property node list in the ONS host and port field when creating a data source. Instead, you should leave this field blank. After you finish creating the data source, you can enter the property node list on the Configuration: ONS tab on the settings page for the data source.

You can specify the following keys in a property node list:

- `nodes.id`—A list of nodes representing a unique topology of remote ONS servers. id specifies a unique identifier for the node list. Duplicate entries are ignored. The list of nodes configured in any list must not include any nodes configured in any other list for the same client or duplicate notifications will be sent and delivered. The list format is a comma separated list of ONS daemon listen addresses and ports pairs separated by colon.

- `maxconnections.id`—Specifies the maximum number of concurrent connections maintained with the ONS servers. id specifies the node list to which this parameter applies. The default is `3`
- `active.id` If `true`, the list is active and connections are automatically established to the configured number of ONS servers. If false, the list is inactive and is only be used as a fail over list in the event that no connections for an active list can be established. An inactive list can only serve as a fail over for one active list at a time, and once a single connection is re-established on the active list, the fail-over list reverts to being inactive. Note that only notifications published by the client after a list has failed over are sent to the fail over list. `id` specifies the node list to which this parameter applies. The default is `true`
- `remotetimeout` —The timeout period, in milliseconds, for a connection to each remote server. If the remote server has not responded within this timeout period, the connection is closed. The default is 30 seconds

> **Note:**
>
> Although `walletfile` and `walletpassword` are supported in the string, WebLogic Server has separate configuration elements for these values, `OnsWalletFile` and O`nsWalletPasswordEncrypted`.

Secure ONS Client Communication

To use an Oracle Wallet file with WebLogic Server, you must:

- Update your Active GridLink data source configuration to include the directory of the Oracle wallet file in which the SSL certificates are stored and optionally, the ONS Wallet password.
- For more information on Oracle Wallet, see the Creating and Managing Oracle Wallet.

Test ONS Client Configuration

`Test ONS client configuration` allows you to test a connection to the ONS listener before the data source configuration is finalized.

Target the Data Source

You can select one or more targets to which to deploy your new Active GridLink data source. If you don't select a target, the data source will be created but not deployed. You will need to deploy the data source at a later time.

Configuring Oracle Parameters

WebLogic Server provides several attributes that provide improved data source performance when using Oracle drivers. See Advanced Configurations for Oracle Drivers and Databases.

Configuring an ONS Client Using WLST

Use WLST to configure an ONS client.

The following fragment provides an example for setting the Oracle parameters of an Active GridLink data source.

```
cd('/JDBCSystemResources/' + dsName + '/JDBCResource/' + dsName + '/
JDBCOracleParams/' + dsName)
cmo.setFanEnabled(true)
cmo.setOnsNodeList('nodes.1=rac1:6200,rac2:6200\nmaxconnections.1=3\n')
```

For more information about configuring an ONS client, see ONS Client Communication.

Configuring Runtime Load Balancing using SDP

To configure load balancing across SDP connections, you must edit the `TNSNAMES.ORA` file on all nodes and add an SDP end-point to the `LISTENER_IBLOCAL` entry.

Note:

The `TNSNAMES.ORA` file is only read at instance startup or when using an `ALTER SYSTEM SET LISTENER_NETWORKS="`*`listener address`*`"` command. After updating the `TNSNAMES.ORA` file, restart all instances or run the `ALTER SYSTEM SET LISTENER_NETWORKS` command on all networks.

For example:

```
LISTENER_IBLOCAL =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST =

   sclcgdb02ibvip.country.myCorp.com)(PORT=1522))
       (ADDRESS = (PROTOCOL = SDP)(HOST =
    sclcgdb02-bvip.country.myCorp.com)(PORT=1522))
    )
  )
```

You should then distribute connections on the `LISTERNER_IB` network using the following URL:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=SDP)  (HOST=sclcgdb01-
bvip.country.myCorp.com)(PORT=1522))(ADDRESS=(PROTOCOL=SDP)  (HOST=sclcgdb02-
ibvip.country.myCorp.com)(PORT=1522)))(CONNECT_DATA=(SERVICE_NAME=elservice)))
```

Configuring Active GridLink Connection Pool Features

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. Connection pools have options that allow you to control JDBC driver features and system properties associated with connection pools as well as use SQL for database connection initialization.

> **Note:**
>
> Certain Oracle JDBC extensions may durably alter a connection's behavior in a way that future users of the pooled connection will inherit. WebLogic Server attempts to protect connections against some types of these calls when possible.

For more information, see *Oracle WebLogic Remote Console Online Help* and JDBCConnectionPoolParamsBean in *MBean Reference for Oracle WebLogic Server*.

The following connection pool options are available for a JDBC data source:

- Enabling JDBC Driver-Level Features
- Enabling Connection-based System Properties
- Initializing Database Connections with SQL Code

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 by 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.

> **Note:**
>
> Do not use `FastConnectionFailoverEnabled`, `ConnectionCachingEnabled`, or `ConnectionCacheName` as Driver-level properties in the `Properties` attribute in a JDBC data source.

Enabling Connection-based System Properties

WebLogic JDBC data sources support setting driver properties using the value of system properties. The value of each property is derived at runtime from the named system property. You can configure connection-based system properties using the WebLogic Remote Console by editing the `System Properties` attribute of your data source configuration.

> **Note:**
>
> Do not specify `oracle.jdbc.FastConnectionFailover` as a Java system property when starting the WebLogic Server.

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 in the WebLogic Remote Console. Alternatively, you can specify simply a table name without `SQL` and

the statement `SELECT COUNT(*) FROM tablename` is used. 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. An Oracle DBMS example:

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

The SQL statement is executed using `JDBC Statement.execute()`. Options that you can set using **InitSQL** vary by DBMS. See the documentation from your database vendor for supported statements. If you want to execute multiple statements, you may want to create a stored procedure and execute it. The syntax is vendor specific. For example, to execute an Oracle stored procedure:

```
SQL CALL MYPROCEDURE()
```

Tuning Active GridLink Data Source Connection Pools

By properly configuring the connection pool attributes in JDBC data sources in your WebLogic Server domain, you can improve application and system performance.

See Tuning Data Source Connection Pools.

Monitoring Active GridLink JDBC Resources

Learn about monitoring and debugging Active GridLink data sources.

For more information, see Monitoring WebLogic JDBC Resources.

- Viewing Run-Time Statistics
- Debug Active GridLink Data Sources

Viewing Run-Time Statistics

You can view run-time statistics for an Active GridLink data source via the *Oracle WebLogic Remote Console Online Help* or through the associated runtime MBeans.

- JDBCOracleDataSourceRuntimeMBean
- JDBCOracleDataSourceInstanceRuntimeMBean
- ONSDaemonRuntimeMBean

JDBCOracleDataSourceRuntimeMBean

The `JDBCOracleDataSourceRuntimeMBean` provides methods for getting the current state of the data source instance and for getting statistics about the data source, such as the average number of active connections, the current number of active connections, and the highest number of active connections. This MBean also has a child `JDBCOracleDataSourceInstanceRuntimeMBean` for each node that is active in the Active GridLink data source. See JDBCOracleDataSourceRuntimeMBean in the *MBean Reference for Oracle WebLogic Server*.

JDBCOracleDataSourceInstanceRuntimeMBean

The `JDBCOracleDataSourceInstanceRuntimeMBean` provides methods for getting the current state of the data source instance. There an instance for each ONS listener that is active. In a configuration that uses `auto-ONS` where the administrator doesn't configure the ONS string, this is the only way to discover which ONS listeners are available. See JDBCOracleDataSourceInstanceRuntimeMBean in the *MBean Reference for Oracle WebLogic Server*.

ONSDaemonRuntimeMBean

The `ONSDaemonRuntimeMBean` provides methods for monitoring the ONS client configuration that is associated with an Active GridLink data source.

The following is a WLST script for testing an ONS connection. In this example, the Active GridLink data source is named **glds** and it is targeted to **myserver**:

```
connect(<wluser>, <wlpassword>, 't3://localhost:7001')
serverRuntime()
cd('JDBCServiceRuntime')
cd('myserver')
cd('JDBCDataSourceRuntimeMBeans')
cd('glds')
cd('ONSClientRuntime')
cd('glds')
cd('ONSDaemonRuntimes')
cd('glds_0')
cmo.ping()
```

See ONSDaemonRuntimeMBean in the *MBean Reference for Oracle WebLogic Server*.

Debug Active GridLink Data Sources

You can activate WebLogic Server's debugging features to track down the specific problem within the application.

- JDBC Debugging Scopes
- UCP JDK Logging
- Enable Debugging Using the Command Line

JDBC Debugging Scopes

The following are registered debugging scopes for JDBC:

- `DebugJDBCRAC`—prints information about Active GridLink data source lifecycle, Universal Connection Pool callback, and connection information.
- `DebugJDBCONS`—traces ONS client information, including the LBA event body. One trace is available for each ONS listener that is active. In a configuration that uses auto-ONS where the administrator doesn't configure the ONS string, this is the only way to see what ONS listeners are available.
- `DebugJDBCReplay`—traces JDBC Replay Driver replay information.
- `DebugJDBCUCP`—traces low level RAC information from the UCP driver.

UCP JDK Logging

For enabling UCP JDK logging, see Overview of Logging in UCP in *Universal Connection Pool for JDBC Developer's Guide*.

Enable Debugging Using the Command Line

Set the appropriate AGL data source debugging properties on the command line. For example,

```
-Dweblogic.debug.DebugJDBCRAC=true
-Dweblogic.debug.DebugJDBCONS=true
-Dweblogic.debug.DebugJDBCUCP=true
-Dweblogic.debug.DebugJDBCREPLAY=true
```

Setting these values is static and can only be used at server startup.

To enable ONS debugging, you must configure Java Util Logging. To do so, set the following properties on the command line as follows:

```
-Doracle.ons.debug=true
```

See java.util.logging in Java Platform Standard Edition API Specification.

Using Active GridLink Data Sources without FAN Notification

You can configure and use an Active GridLink data source without enabling Fast Application Notification (FAN). In this configuration, disabling a connection to a RAC node occurs after two successive connection test failures. Connectivity is reestablished after a successful connection test.

Note:

This is not a standard recommendation from Oracle.

Oracle recommends that you enable `TestConnectionsOnReserve`. You might need to turn off FAN if a configured firewall doesn't allow this protocol to flow.

The following table indicates the availability of Active GridLink data source features when FAN Enabled set to `false`.

Table 4-2 Active GridLink Features when FAN Enabled is False

Active GridLink Feature	Available when FAN Enabled is False?
Single data source configuration for access to RAC cluster	Yes
Runtime MBeans for individual RAC cluster instances	Yes
Connection load balancing using Runtime Load Balancing (RLB)	No
Fast Application Notification (FAN)	No
Fast Connection Failover (FCF)	No

Table 4-2 (Cont.) Active GridLink Features when FAN Enabled is False

Active GridLink Feature	Available when FAN Enabled is False?
Graceful shutdown	No
Gravitation (rebalancing connections)	No
ONS Client Support, including password and encrypted wallet configurations	Yes
Transaction affinity	Yes
Session affinity	No

Understanding the ActiveGridlink Attribute

In WebLogic Server 12.1.2 and higher, the `ActiveGridlink` attribute is used to explicitly declare a data source configuration as an Active GridLink data source. It is automatically enabled by the WebLogic Remote Console when creating a Active GridLink data source. If you create data source configurations using WLST, you must remember to set `ActiveGridlink=true`.

Note:

To maintain backward compatibility with releases prior to WebLogic Server 12.1.2, a data source configuration is always an Active GridLink data source configuration if `FanEnabled=true` or the `OnsNodeList` is non-null. In this case, the ActiveGridlink value is ignored.

Legacy data source configurations are not updated during the upgrade process. If you need to update a legacy Active GridLink data source to access RAC clusters without enabling Fast Application Notification (FAN), edit or use WLST to set `ActiveGridlink=true` in the configuration.

Best Practices for Active GridLink Data Sources

Learn about the best practices for using Active GridLink data sources by understanding the catch and handle exceptions and how connections are created when using an Active GridLink data source.

- Catch and Handle Exceptions
- Connection Creation with Active GridLink Data Sources

Catch and Handle Exceptions

Applications need to catch and handle all exceptions. Applications using Active GridLink data sources should expect exceptions, such as an `IO socket read error`, when performing JDBC operations on borrowed connections. Best practice is to check the connection validity and reconnect if necessary. Connection exceptions can occur if the driver detects an outage earlier than FAN event arrival or as a result of the cleanup of a connection. For unplanned down events, a connection pool aborts all borrowed connections that are affected by the outage.

Connection Creation with Active GridLink Data Sources

This section summarizes the change in connections in Active GridLink data source, assuming FAN and ONS are enabled:

- Connections are added to the pool initially based on the configured initial capacity. That uses connect time load balancing based on the listener. For that to work correctly, you must either specify `LOAD_BALANCE=ON` for multiple non-scan addresses or use SCAN.
- Connections are added to the pool on demand based on runtime load balancing. However, this is overridden by XA affinity or Web session affinity, in which case connections are added on the instance providing affinity to the last request in the transaction or Web session.
- When a planned down event occurs, unused connections for that instance are released immediately and connections in use are released when returned to the pool.
- When an unplanned down event occurs, all connections for that instance are destroyed immediately.
- When an up event occurs, connections are proactively created on the new instance.
- When gravitation shrinking occurs, one unused connection is destroyed on a heavily loaded instance (per period).
- When normal shrinking occurs, half of the unused connections down to minimum capacity are destroyed without respect to load (per period).

Comparing Active GridLink and Multi Data Sources

There are several benefits to using Active GridLink data sources over Multi Data Sources when using Oracle RAC clusters.

The benefits include:

- Requires one data source with a single URL. Multi Data Sources require a configuration with *n* Generic data sources and a Multi Data Source.
- Eliminates a polling mechanism that can fail if one of the Generic data sources is performing slowly.
- Eliminates the need to manually add or delete a node to/from the cluster.
- Provides a fast internal notification (out-of-band) when nodes are available so that connections are load-balanced to the new nodes using Oracle Notification Service (ONS).
- Provides a fast internal notification when a node goes down so that connections are steered away from the node using ONS.
- Provides load balancing advisories (LBA) so that new connections are created on the node with the least load, and the LBA information is also used for gravitation to move idle connections around based on load.
- Provides affinity based on your XA transaction or your web session which may significantly improve performance.
- Leverages all the advantages of HA configurations like DataGuard. For more information, see Oracle WebLogic Server and Highly Available Oracle Databases: Oracle Integrated Maximum Availability Solutions on the Oracle Technology network at http://www.oracle.com/technetwork/middleware/weblogic/learnmore/index.html.

Migrating from Multi Data Source to Active GridLink

You can migrate to Multi Data Source from Active GridLink data sources using simple manual process.

- Application Changes to Migrate a Multi Data Source
- Configuration Changes to Migrate a Multi Data Source
- Basic Migration Steps

Application Changes to Migrate a Multi Data Source

No changes should be required to your applications. A standard application looks up the Multi Data Source in JNDI and uses it to get connections. By giving the Active GridLink data source the same JNDI name as the Multi Data Source, the process is exactly the same in the application to use a data source name from JNDI.

Configuration Changes to Migrate a Multi Data Source

The only changes necessary should be to your configuration. An Active GridLink data source (AGL) is composed of information from the Multi Data Source (MDS) and the member generic data sources combined into a single AGL descriptor. The only additional information that is needed is the configuration of Oracle Notification Service (ONS) on the RAC cluster. In many cases, the ONS information consists of the same host names as used in the MDS and the only additional information is the port number, and which can be simplified by the use of a SCAN address.

A MDS descriptor does not contain much information. The key components are:

- The JNDI name. It must become the name of your new AGL data source to keep things transparent to the application. If you want to run the MDS in parallel with the AGL data source, then you must give the AGL data source a new JNDI name but you must also update the application to use that new JNDI name.
- A list of the member Generic data sources which provide any remaining information that you need to configure the AGL data source.

  Each of the member Generic data sources has its own URL. As described in Using Multi Data Sources with Oracle RAC, it has the following pattern:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=
       (PROTOCOL=TCP)(HOST=host1-vip)(PORT=1521))
       (CONNECT_DATA=(SERVICE_NAME=dbservice)(INSTANCE_NAME=inst1)))
  ```

  Each member should have its own host and port pair. The members probably have the same service and often have the same port on different hosts. The URL for the AGL data source is a combination of the host and port pairs. For example:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=
      (ADDRESS=(PROTOCOL=TCP)(HOST=host1-vip)(PORT=1521))
      (ADDRESS=(PROTOCOL=TCP)(HOST=host2-vip)(PORT=1521)))
      (CONNECT_DATA=(SERVICE_NAME=dbservice))
  ```

  It is preferable to use an Oracle Single Client Access Name (SCAN) address instead of multiple host or Virtual IP (VIP) addresses. SCAN addresses are simpler and makes changes to the nodes in the cluster transparent. For more information on SCAN addresses, see the *Oracle Real Application Clusters Administration and Deployment Guide*. For example:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
(HOST=scanaddress)(PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=dbservice))
```

- Ignore the **Algorithm Type**.

Basic Migration Steps

The following section provides the basic steps needed to migrate a Multi Data Source to an Active GridLink data source:

- Delete the Multi Data Source and the Generic data sources from the configuration using the WebLogic Remote Console.
- Add a single Active GridLink data source using the WebLogic Remote Console.
  - Give it the same JNDI name as the Multi Data Source.
  - Select an XA or non-XA driver based on your what Generic data sources used.
  - Enter the complete URL as described in Configuration Changes to Migrate a Multi Data Source.
  - Set the user and password, it should be the same as what you had on the Multi Data Source members.
  - On the **Test GridLink Datasource Connection** page, click **Test All Listeners** and verify the new URL.
  - Enter the information for the ONS connections. Specify one or more *host*:*port* pairs. For example, `host1-vip:6200` or `scanaddress:6200`. If possible, use a single SCAN address and port. Make sure that **FAN Enabled** is checked.
  - Test the ONS connections.
- Deploy the data source.
- Edit the Active GridLink data source and configure additional parameters.

  There are many data source parameters that can't be configured while creating a new data source. In most cases, you should be able to use the parameter setting used in the Multi Data Source. If there are conflicts, you will need to resolve them and select the appropriate settings for your environment.

For more information on creating Active GridLink data sources using the WebLogic Remote Console, see *Oracle WebLogic Remote Console Online Help*.

Managing Database Downtime with Active GridLink Data Sources

Learn several ways to handle database downtime with Active GridLink data sources in an Oracle RAC database environment.

- Active GridLink Configuration for Database Outages
- Planned Outage Procedures
- Unplanned Outages

Active GridLink Configuration for Database Outages

Ensure that the Active GridLink data source is configured as follows:

- Fast Application Notification (FAN) is enabled. FAN provides rapid notification about state changes for database services, instances, the databases themselves, and the nodes that

form the cluster. It allows for draining of work during planned maintenance with no errors returned to applications.

- Is using auto-ONS, or an explicitly defined ONS configuration. See ONS Client Configuration.
- Is using a dynamic database service. Do not connect using the administrative service or PDB service. They are for intended for administration purposes only and are not supported for FAN.
- Test connections is enabled. Depending on the outage, applications may receive stale connections when connections are borrowed before a down event is processed. This can occur, for example, on a clean instance down when sockets are closed coincident with incoming connection requests. To prevent the application from receiving any errors, connection checks should be enabled at the connection pool. This requires setting `test-connections-on-reserve` to true and setting the `test-table` (the recommended value for Oracle is `SQL ISVALID`).
- SCAN usage is optimized. For database drivers 12.1.0.2 and later, set the URL setting `LOAD_BALANCE=TRUE` for the `ADDRESSLIST` as an optimization to force re-ordering of the SCAN IP addresses that are returned from DNS for a SCAN address.

For database drivers before 12.1.0.2, use the connection property `oracle.jdbc.thinForceDNSLoadBalancing=true`. See SCAN Addresses.

Planned Outage Procedures

For planned downtime, the primary goal is to manage scheduled maintenance with no application interruption while maintenance is underway at the database server. Achieving this goal requires the following:

- Transparent scheduled maintenance—Ensures that the scheduled maintenance process at the database servers is transparent to applications.
- Session Draining—When an instance is brought down for maintenance at the database server, session draining ensures that all work using instances at that node completes and that idle sessions are removed. Sessions are drained without impacting in-flight work.

For maintenance purposes (such as software and hardware upgrades, repairs, changes, migrations within and across systems), the services used are shutdown gracefully one or several at a time without disrupting the operations and availability of the WebLogic Server applications. Upon a FAN DOWN event, Active GridLink drains sessions away from the instance(s) targeted for maintenance. It is necessary to stop non-singleton services running on the target database instance (assuming that they are still available on the remaining running instances) or relocate singleton services from the target instance to another instance. Once the services have drained, the instance is stopped with no application errors

The following steps provide a high level overview of the planned maintenance process:

1. Detect `DOWN` event triggered by DBA on instances targeted for maintenance.
2. Drain sessions away from the targeted instance(s).
3. Perform scheduled maintenance on the database servers.
4. Resume operations on the upgraded node(s).

Unlike Multi Data Source where operations need to be coordinated on both the database server and the mid tier, Active GridLink co-operates with the database so that all of these operations are managed from the database server, simplifying the process. Table 4-3 lists the steps that are executed on the database server and the corresponding reactions at the mid tier.

Table 4-3 Steps Performed on Database Server for Active GridLink Planned Maintenance

Step #	Database Server Steps	Command	Mid-Tier Reaction

1.	Stop the non-singleton service without -force or relocate the singleton service.
	Omitting the –server option operates on all services on the instance.
		`$ srvctl stop service -db db_name -service service_name -instance instance_name`
		or
		`$ srvctl relocate service -db db_name -service service_name -oldinst oldins -newinst newinst`
			The FAN Planned Down (`reason=USER`) event for the service informs the connection pool that a service is no longer available for use and connections should be drained. Idle connections on the stopped service are released immediately. In-use connections are released when returned (logically closed) by the application. New connections are reserved on other instance(s) and databases offering the services. This FAN action invokes draining the sessions from the instance without disrupting the application.

2.	Disable the stopped service to ensure it is not automatically started again. Disabling the service is optional. This step is recommended for maintenance actions where the service must not restart automatically until the action has completed.	`$ srvctl disable service -db db_name -service service_name -instance instance_name`	No new connections are associated with the stopped/disabled service at the mid-tier.
3.	Allow sessions to drain.	Not applicable	The amount of time depends on the application. There may be long-running queries. Batch programs may not be written to periodically return connections and get new ones. It is recommended that batch be drained in advance of the maintenance.

Table 4-3 (Cont.) Steps Performed on Database Server for Active GridLink Planned Maintenance

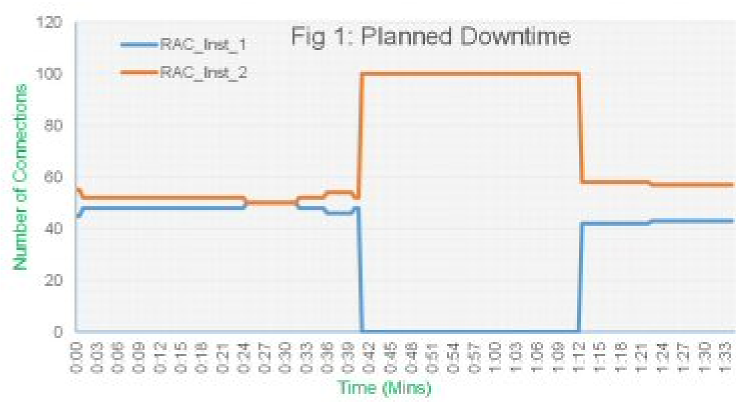
Step #	Database Server Steps	Command	Mid-Tier Reaction
4.	Check for long-running sessions. Terminate these sessions using a transactional disconnect. Wait for the sessions to drain. You can run the query again to check if any sessions remain.	`SQL> select count(*) from (select 1 from v$sessionwhere service_name in upper('service_name') union all select 1 from v$transaction where status = 'ACTIVE' )` `SQL> exec dbms_service.disconnect_session( 'service_name', DBMS_SERVICE.POST_TRANSACTION)`	The connection on the mid-tier will get an error. If using JDBC Replay Driver, it is possible to hide the error from the application by automatically replaying the operations on a new connection on another instance. Otherwise, the application gets a SQLException.
5.	Repeat steps 1 through 4.	Repeat for all services targeted for planned maintenance	Not Applicable
6.	Stop the database instance using the immediate option.	`$ srvctl stop instance -db db_name -instance instance_name -stopoption immediate`	No impact on the mid-tier until the database and service are re-started.
7.	Optionally, disable the instance so that it will not automatically start again during maintenance. This step is for maintenance operations where the services cannot resume during the maintenance.	`$ srvctl disable instance -db db_name -instance instance_name`	Not Applicable
8.	Perform the scheduled maintenance work (patches, repairs, and changes).	Not Applicable	Not Applicable

Table 4-3 (Cont.) Steps Performed on Database Server for Active GridLink Planned Maintenance

Step #	Database Server Steps	Command	Mid-Tier Reaction
9.	Enable and start the instance.	`$ srvctl enable instance -db db_name -instance instance_name` `$ srvctl start instance -db db_name -instance instance_name`	Not Applicable
10.	Enable and start the service back. Check that the service is up and running.	`$ srvctl enable service -db db_name -service service_name -instance instance_name` `$ srvctl start service -db db_name -service service_name -instance instance_name`	The FAN UP event for the service informs the connection pool that a new instance is available for use, allowing sessions to be created on this instance at the next request submission. Automatic rebalancing of sessions starts.

The following figure shows the distribution of connections for a service across two Oracle RAC instances before and after Planned Downtime. Notice that the connection workload moves from fifty-fifty across both instances to hundred-zero. In other words, RAC_INST_1 can be taken down for maintenance without any impact on the business operation.

Figure 4-6 Distribution of Connections Across Two Oracle RAC Instances



Unplanned Outages

There are several differences when an unplanned outage occurs:

- A component at the database server may fail making all services unavailable on the instances running at that node. There is no stop or disable on the services because they have failed.
- The FAN unplanned DOWN event (`reason=FAILURE`) is delivered to the mid-tier.
- All sessions are closed immediately, preventing the application from hanging on TCP/IP timeouts. Existing connections on other instances remain usable, and new connections are opened to these instances as needed.
- There is no graceful draining of connections. For those applications using services that are configured to use JDBC Replay Driver, active sessions are restored on a surviving instance and recovered by replaying the operations, masking the outage from applications. If not protected by JDBC Replay Driver, any sessions in active communication with the instance receive a `SQLException`.

Gradual Draining

During planned database maintenance, gradually close the database connections instead of closing all of the connections immediately. This strategy prevents uneven performance by the application.

When planned database maintenance occurs, a planned down service event is processed by the WebLogic Server JDBC data source. By default, all unreserved connections in the pool are closed immediately and borrowed connections are closed when they are returned to the pool. This shutdown process can cause uneven application performance because:

- New connections need to be created on the alternative instances.
- A logon storm can occur on the other instances.

This feature is supported for an Active GridLink data source running with Oracle RAC.

Setting the Drain Timeout Period

The connection property `weblogic.jdbc.drainTimeout` is recognized to define the draining period in seconds. The value must be a non-negative integer. For example, the following is a sample from a WLST script that creates a data source.

```
jdbcSR = create(dsname, 'JDBCSystemResource')
jdbcResource = jdbcSR.getJDBCResource()
driverParams = jdbcResource.getJDBCDriverParams()
driverProperties = driverParams.getProperties()
drainprop = driverProperties.createProperty('weblogic.jdbc.drainTimeout')
drainprop.setValue('60')
```

When running with the Oracle database 12.2 driver and the Oracle database 12.2 server, the drain timeout can be configured on the database server side by setting `-drain_timeout` on the database service. For example, a repayable service can be created by using:

```
srvctl add service -db ORCL -service otrade -clbgoal SHORT  -preferred
orcl1,orcl2 -rlbgoal SERVICE_TIME -failoverretry 30 -failoverdelay 10  -
failovertype TRANSACTION -commit_outcome TRUE -replay_init_time 1800 -retention
86400 -notification TRUE -drain_timeout 60
```

If both the connection property and the server-side drain timeout are set on an Oracle database 12.2 configuration, the server-side value takes precedence. This value is only used during a planned down event to stop some but not all of the instances on which a service is running. For example,

```
srvctl stop service -db ORCL -instance orcl2 -service otrade.example.com
```

If the drain period is not set or set to 0, then by default, there is no drain period and connections are closed immediately.

A small value accelerates the migration, but might cause applications to experience higher response times, as requests on the target node hit a cold buffer cache. A larger value migrates work more gently and gives the buffer cache on the target node more time to warm-up, which in consequence leads to reduced impact on the application, but a longer overall migration duration.

Gradual Draining Processing

Processing starts when a database service that is configured for an Active GridLink data source is stopped using `srvctl stop service -db dbname -instance instancename -service servicename`.

> **Note:**
>
> Draining is not done if all services are shutdown (for example, when no instance name is specified).

- If the drain timeout is not set or set to 0, there is no drain period. Unreserved connections are immediately closed and borrowed connections are closed when returned to the pool.
- If the drain timeout is specified, it takes effect only if the service is available at another RAC instance. For active/active services draining is gradual. For active/passive services, version 12.2 of RAC relocates the service first, so gradual draining is also supported. This feature does not work with Oracle DataGuard, which has only one primary active service at a time.
- If an alternative instance is available, the drain timeout period is started. The granularity and reducing the connections is done on a five-second interval. The total connection count is the count of the unreserved and the count of the reserved connections. The total count is divided by the value “(drain period/5)” to compute the number of connections to be released per interval (note that if the number is less than 1, then some intervals may not have any connections drained). After each five-second interval, harvestable connections are harvested and interval count connections are closed if they are unreserved or marked for closure on return to the pool. After the last interval, the instance is marked as down (with respect to monitor status).
- If a data source is suspended or shut down, draining is stopped on any instance that is currently draining. Unreserved connections are immediately closed and borrowed connections are closed when returned to the pool.
- If a service is started again on an instance that is draining for that service, draining is stopped.
- If a service is stopped on all instances by not specifying a instance name or the last instance is stopped, draining is stopped on all instances. For all instances, unreserved connections are immediately closed and borrowed connections are closed when returned to the pool.

- When draining is happening on an instance, connection gravitation on the data source (rebalancing connections based on the runtime load balancing information) is stopped until the draining completes.
- When the service is stopped, the Load Balance Advisories (LBA) indicates that the percentage for the stopped service should be 0. This causes the preference for allocating existing connections to other instances first. If a connection does not exists on the other instances and a connection exists on the stopped service, it will pick that one instead of creating a connection. This applies to connections created using LBA or Session Affinity. XA affinity will try to create a new connection for the instance in the affinity context, and only use a different instance or branch if a new connection can't be created.

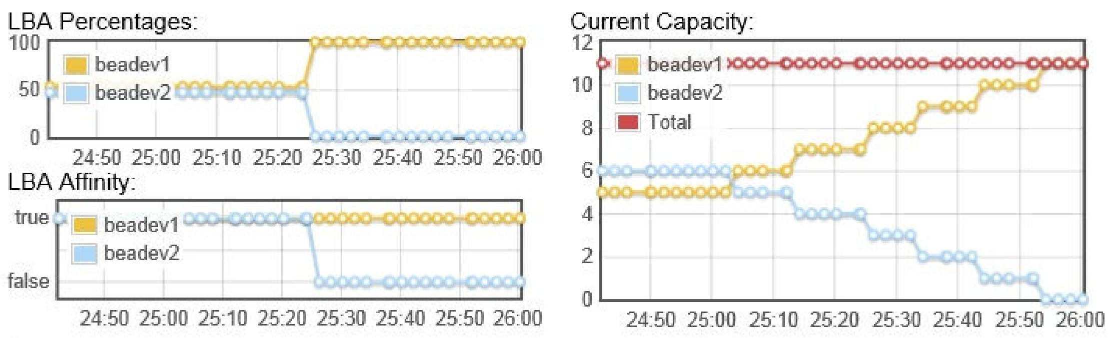
Example

The following figure shows the effect of gradual draining when a service on an instance is stopped. In this case, the service is stopped on instance *`beadev2`* just after 25:00. Note that it takes a while for the Load Balancing Advisories (LBA) to respond to the shut down at around `25:25` and the percentage goes to `0` for instance *`beadev2`*. WebLogic Server receives the shutdown event almost instantly and starts to take action. If gradual draining were not configured, the graph of *Current Capacity* would show the capacity dropping to `0` (or the count of active connections) immediately when the event is received. Instead, you can see that the capacity gradually goes down every five- seconds for the sixty-second drain period and there is a corresponding increase in capacity on *`beadev1`*. Note that the total capacity stays constant through the entire period.

Note:

These graphs were generated from an artificial work-load of requests that are getting a connection, doing a little work, and releasing the connection. In the real world, the results may not be so perfect.

Figure 4-7 Gradual Draining



Using Universal Connection Pool Data Sources

A Universal Connection Pool (UCP) data source is provided as an option for users who wish to use Oracle Universal Connection Pooling to connect to Oracle Databases. UCP provides an alternative connection pooling technology to Oracle WebLogic Server connection pooling.

- What is Universal Connection Pool Data Source
- Creating a Universal Connection Pool Data Source
  To create a Universal Connection Pool data source in your WebLogic domain, you can use the WebLogic Remote Console, the WebLogic Scripting Tool (WLST), or Fusion Middleware Control.
- Universal Connection Pool Multi Tenant Shared Pool support
  To use this feature, the URI for the Universal Connection Pool (UCP) MT Shared Pool support XML configuration file must be specified using the `oracle.ucp.jdbc.xmlConfigFile` system property before any UCP data source is loaded in the JVM.
- Monitoring Universal Connection Pool JDBC Resources
  Learn about monitoring Universal Connection Pool JDBC Resources using the WebLogic Remote Console or the `JDBCUCPDataSourceRuntimeMBean`, `JDBCDataSourceRuntimeMBean` .
- Oracle Sharding Support
  Sharding is a data tier architecture in which data is horizontally partitioned across independent databases.

What is Universal Connection Pool Data Source

A Universal Connection Pool data source is provided as an option for users who wish to use UCP for connecting to Oracle Databases. UCP provides an alternative connection pooling technology to Oracle WebLogic Server connection pooling.

Note:

Oracle generally recommends the use of Generic data source, Multi Data Source, or Active GridLink data source with Oracle WebLogic Server to establish connectivity with Oracle databases.

WebLogic Server provides the following support when using a UCP data source:

- Configuration as an alternative data source to Generic data source, Multi Data Source, or Active GridLink data source.
- Deploy and undeploy data source.
- Basic monitoring and statistics:
  - `ConnectionsTotalCount`
  - `CurrCapacity`
  - `FailedReserveRequestCount`
  - `ActiveConnectionsHighCount`
  - `ActiveConnectionsCurrentCount`
- Certification with Oracle simple driver, XA driver, and JDBC Replay Driver driver.

A UCP data source does not support:

- Additional life cycle operations (suspend, resume, shutdown, forceshutdown, start, and so on).
- Generic support for any connection pool.

- Oracle WebLogic Server Security options.
- JDBC drivers other than those listed above.
- RMI access to a UCP data source.
- Integration with the WebLogic Transaction Manager (OnePhaseCommit, EmulateTwoPhaseCommit, LoggingLastResource and TwoPhaseCommit).
- Oracle WebLogic Server data operations such as JMS, Leasing, EJB, JDBC TLOG, and so on.

The implementations of UCP data sources are loosely coupled, allowing the swapping of the `ucp.jar` to support the use of new UCP features by the applications. UCP data sources are not supported in an application-scoped/packaged or stand-alone module environment.

For details about the Oracle Universal Connection Pool, see Oracle Universal Connection Pool for JDBC Developer's Guide.

Creating a Universal Connection Pool Data Source

To create a Universal Connection Pool data source in your WebLogic domain, you can use the WebLogic Remote Console, the WebLogic Scripting Tool (WLST), or Fusion Middleware Control.

Procedures for creating a Universal Connection Pool data source using Fusion Middleware Control are described in Create JDBC Universal Connection Pool data sources in Administering Oracle WebLogic Server with Fusion Middleware Control.

The WebLogic Remote Console and WLST methods are described in the following sections:

- Configuring a UCP Data Source in the WebLogic Remote Console
- Configuring a UCP Using WLST

Configuring a UCP Data Source in the WebLogic Remote Console

The procedure for creating a Universal Connection Pool (UCP) data source in the WebLogic Remote Console is provided in *Oracle WebLogic Remote Console Online Help*. This procedure includes instructions for accessing the data source configuration wizard.

The following sections provide an overview of the basics steps used in the data source configuration wizard to create a data source using the WebLogic Remote Console:

Set JDBC Data Source Properties

The JDBC Data Source Properties section includes options that determine the identity of the data source and the way the data is handled on a database connection. Guidelines for configuring these properties are described as follows:

- **Data Source Names**—Enter a name for the UCP data source in the Name field. 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. To avoid naming conflicts, data source names should also be unique among other configuration object names, such as servers, applications, clusters, and JMS queues, topics, and servers. The data source name cannot contain the following special characters: `@ # $`.
- **Scope**—Select the scope for the data source from the list of available scopes. You can set the scope to Global (at the domain level), or to any existing Resource Group or Resource Group Template.

- **JNDI Names**—Enter a JNDI name for the UCP data source in the JNDI Name field. You can configure a data source so that it binds to the JNDI tree with a single name or 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. For more information, see Developing JNDI Applications for Oracle WebLogic Server.
- **Database Type and Driver**—The UCP data source is certified with three Oracle drivers: thin XA and non-XA, and an JDBC Replay Driver driver. Select the required driver from the menu.

The supported combinations of driver and JDBC connection factory are shown in Table 4-4

Table 4-4 Supported Driver and Connection Factory Combinations for UCP Data Source

Driver	Factory (ConnectionFactoryClassName)
`oracle.ucp.jdbc.PoolDataSourceImpl (default)`	`oracle.ucp.jdbc.PoolDataSourceImpl`
`oracle.ucp.jdbc.PoolXADataSourceImpl`	`oracle.jdbc.xa.client.OracleXADataSource`
`oracle.ucp.jdbc.PoolDataSourceImpl`	`oracle.jdbc.replay.OracleDataSourceImpl`

Note:

The JDBC Replay Driver does not currently support XA transactions.

If a non-XA driver from the list in Table 4-4 is specified with an XA factory from the table, an error is generated. If you specify values that are not in the table they are not validated.

If the `driver-name` is not specified in the `jdbc-driver-params`, it defaults to `oracle.ucp.jdbc.PoolDataSourceImpl`.

If you specify a supported driver name but do not specify the `ConnectionFactoryClassName` connection property, the corresponding entry from Table 4-4 is used. If you do not specify a supported driver name, an error is generated.

Set Connection Properties

Connection properties are used to configure the connection between the data source and the DBMS. There are two ways that you can enter the connection properties for a UCP data source in the Remote Console.

On the Connection Properties page of the wizard, all of the available connection properties for a UCP driver are displayed so that you can enter the appropriate values. As an alternative, you can configure properties by entering the properties directly into the Properties text box on the Test Database Connection page using the format `propertyName=value`. Any values that you entered on the Connection Properties page are already listed Properties text box.

Table 4-5 describes the connection properties that you can configure for a UCP data source. For more information about UCP properties, see Class PoolDataSourceImpl. In Oracle Universal Connection Pool for JDBC Java API Reference. Attributes are determined by setters

on the PoolDataSourceImpl class. Use the attribute name without the "set" prefix. The names are case insensitive.

Table 4-5 Universal Connection Pool Properties

Property	Type	Description
AbandonedConnectionTimeout	int	Sets the abandoned connection timeout. The range of valid values is 0 to Integer.MAX_VALUE. The default is 0.
ConnectionFactoryClassName	String	Sets the Connection Factory class name.
ConnectionFactoryProperties	name=value	Sets the connection factory properties on the connection factory.
ConnectionFactoryProperty	name=value	Sets a connection factory property on the connection factory.
ConnectionHarvestMaxCount	int	Sets the maximum number of connections that can be harvested when the connection harvesting occurs.
ConnectionHarvestTriggerCount	int	Sets the number of available connections at which the connection pool's connection harvesting will occur.
ConnectionLabelingHighCost	int	Sets the cost value that identifies a connection as "high-cost" for connection labeling.
ConnectionPoolName	String	Sets the connection pool name.
ConnectionProperties	name=value	Sets the connection properties on the connection factory.
ConnectionProperty	name=value	Sets a connection property on the connection factory.
ConnectionWaitTimeout	int	Sets the amount of time to wait (in seconds) for a used connection to be released by a client. The range of valid values is 0 to Integer.MAX_VALUE. The default is 3.
DatabaseName	String	Sets the database name.
DataSourceName	String	Sets the data source name.
Description	String	Sets the data source description.
FastConnectionFailoverEnabled	Boolean	Enables Fast Connection Failover (FCF) for the connection pool accessed using this pool-enabled data source. Valid values are true and false.

Table 4-5 (Cont.) Universal Connection Pool Properties

Property	Type	Description
`HighCostConnectionReuseThreshold`	`int`	Sets the high-cost connection reuse threshold for connection labeling.
`InactiveConnectionTimeout`	`int`	Sets the inactive connection timeout. he range of valid values is 0 to `Integer.MAX_VALUE`. The default is 0.
`InitialPoolSize`	`int`	Sets the initial pool size. The range of valid values is 0 to `Integer.MAX_VALUE`. It is illegal to set this to a value greater than the maximum pool size. The default is 0.
`LoginTimeout`	`int`	Sets the login timeout.
`MaxConnectionReuseCount`	`int`	Sets the connection reuse count property.
`MaxConnectionReuseTime`	`long`	Sets the connection reuse time property.
`MaxIdleTime`	`int`	Sets Idle timeout for available connections in the pool.
`MaxPoolSize`	`int`	Sets the maximum number of connections. The range of valid values is 1 to `Integer.MAX_VALUE`. The default is `Integer.MAX_VALUE`.
`MaxStatements`	`int`	Sets the maximum number of statements that may be pooled or cached on a connection.
`MinPoolSize`	`int`	Sets the minimum number of connections. The range of valid values is 0 to `Integer.MAX_VALUE`. It is illegal to set this to a value greater than the maximum pool size. The default is 0.
`NetworkProtocol`	`String`	Sets the data source network protocol.
`ONSConfiguration`	`String`	Sets the configuration string used for remote ONS subscription.
`Password`	`String`	Sets the password with which connections have to be obtained.
`PortNumber`	`int`	Sets the database port number.
`PropertyCycle`	`int`	Sets the Property cycle in seconds.
`RoleName`	`String`	Sets the data source role name.

Table 4-5 (Cont.) Universal Connection Pool Properties

Property	Type	Description
ServerName	String	Sets the database server name.
SQLForValidateConnection	String	Sets the value (SQL) for SQLForValidateConnection property.
TimeoutCheckInterval	int	Sets the timeoutCheckInterval, in seconds.
TimeToLiveConnectionTimeout	int	Sets the maximum time, in seconds, that a connection may remain in-use.
URL	String	Sets the URL that the data source uses to obtain connections to the database.
User	String	Sets the user name with which connections have to be obtained.
ValidateConnectionOnBorrow	Boolean	Sets whether or not a connection being borrowed should first be validated. Valid values are true and false.

> **Note:**
>
> System properties and encrypted properties are supported in addition to normal string literals.

If the jdbc-driver-params URL is set, any URL property is ignored. If the encrypted-password is set, any password property is ignored.

The attributes ConnectionFactoryProperty, ConnectionFactoryProperties, ConnectionProperty, and ConnectionFactoryProperties accept values of the form "name1=value1,name2=value2...".

Test Database Connections

The Test Database Connection page allows you to enter free-form values for properties and to test a database connection before the data source configuration is finalized using a table name or SQL statement. If necessary, you can test additional configuration information using the Properties and System Properties attributes.

Select Targets

You can select one or more targets to which to deploy your new UCP data source. If you don't select a target, the data source will be created but not deployed. You will need to deploy the data source at a later time.

Configuring a UCP Using WLST

You can create a UCP data source using WebLogic Scripting Tool (WLST) in the same way that you create other data source types. However, UCP data sources have less attributes.

The configuration elements for a UCP data source are as follows.

- `name`
- `datasource-type=UCP`
- `jdbc-driver-params url`
- `jdbc-driver-params property - user`
- `jdbc-driver-params password-encrypted`
- `jdbc-data-source-params jndi-name`
- `jdbc-driver-params other properties`

No other elements from the WebLogic Server data source descriptor are recognized. If other elements are specified, they are ignored.

A sample WLST script for creating a UCP data source is provided in Example 4-2

Example 4-2 Sample WLST Script to Create a UCP Data Source

```
import sys, socket
import os
hostname = socket.gethostname()
connect("username","password","t3://"+hostname+":7001")
edit()
startEdit()
serverName="AdminServer"
serverBean = getMBean('/Servers/'+serverName)
host='%s.us.example.com' %hostname
print 'Creating UCP datasource'
domain = getMBean("/")
startEdit()
resourceName='ucpDS'
print "Creating datasource ds in domain"
systemResource=domain.createJDBCSystemResource(resourceName)
systemResource.setName(resourceName)
jdbcResource=systemResource.getJDBCResource()
jdbcResource.setName(resourceName)
jdbcResource.setDatasourceType('UCP')
driverParams=jdbcResource.getJDBCDriverParams()
driverParams.setDriverName('oracle.ucp.jdbc.PoolDataSourceImpl')
driverParams.setUrl('jdbc:oracle:thin:@dbhost:1521/otrade')
properties = driverParams.getProperties()
properties.createProperty('user', 'dbuser')
properties.createProperty('ConnectionFactoryClassName',
'oracle.jdbc.pool.OracleDataSource')
driverParams.setPassword('PASSWD')
jdbcDataSourceParams=jdbcResource.getJDBCDataSourceParams()
jdbcDataSourceParams.addJNDIName(resourceName)
jdbcDataSourceParams.setGlobalTransactionsProtocol('None')
cd('/SystemResources/' + resourceName )
```

```
set('Targets',jarray.array([ObjectName('com.bea:Name=' + serverName +
',Type=Server')], ObjectName))
save()
activate()
```

Note:

You can also use the sample WLST script for creating a Generic data source that is provided with WebLogic Server as the basis for your UCP data source:

```
EXAMPLES_HOME\wl_server\examples\src\examples\wlst\online\jdbc_data_so
urce_creation.py
```

where `EXAMPLES_HOME` represents the directory in which the WebLogic Server code examples are configured. See WLST Online Sample Scripts in *Understanding the WebLogic Scripting Tool*.

Universal Connection Pool Multi Tenant Shared Pool support

To use this feature, the URI for the Universal Connection Pool (UCP) MT Shared Pool support XML configuration file must be specified using the `oracle.ucp.jdbc.xmlConfigFile` system property before any UCP data source is loaded in the JVM.

This can be set on the command line when starting Weblogic Server. Since this is sometimes inconvenient, it is also possible to set the `XmlConfigFile` connection property. If you use the connection property approach, it must be set on all UCP data sources configured in WebLogic Server, even if they do not use the XML file. The format is generally something like `file:///path/file.xml`.

See Overview of UCP Shared Pool for Database Sharding in *Universal Connection Pool Developer's Guide* .

When using the shared pool feature, all attributes for the data source are ignored except for the following:

- Name – the data source name
- Data source Type – UCP
- Driver class name – `oracle.ucp.jdbc.PoolDataSourceImpl` or `oracle.ucp.jdbc.PoolXADataSourceImpl`
- Property `DataSourceFromConfiguration` – data source name in the XML file
- Property `XmlConfigFile` – optionally set the URI of the XML file if not set as a system property
- JNDI Name – the JNDI name where the data source object is mapped

  Example:

  ```
  import sys, socket
  import os
  hostname = socket.gethostname()
  connect("weblogic","server_password","t3://"+hostname+":7001")
  edit()
  ```

```
startEdit()
serverName="myserver"
print 'Creating UCP datasource'
domain = getMBean("/")
startEdit()
resourceName='ds5'
print "Creating datasource ds in domain"
systemResource=domain.createJDBCSystemResource(resourceName)
systemResource.setName(resourceName)
jdbcResource=systemResource.getJDBCResource()
jdbcResource.setName(resourceName)
jdbcResource.setDatasourceType('UCP')
driverParams=jdbcResource.getJDBCDriverParams()
driverParams.setDriverName('oracle.ucp.jdbc.PoolDataSourceImpl')
properties = driverParams.getProperties()
properties.createProperty('DataSourceFromConfiguration', 'pds1')
properties.createProperty('XmlConfigFile', 'file:///SharedPoolDemo.xml')
jdbcDataSourceParams=jdbcResource.getJDBCDataSourceParams()
jdbcDataSourceParams.addJNDIName(resourceName)
cd('/SystemResources/' + resourceName )
set('Targets',jarray.array([ObjectName('com.bea:Name=' + serverName +
',Type=Server')], ObjectName))
save()
activate()
```

The UCP XML file might look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<ucp-properties>
<connection-pool
connection-pool-name="pool1"
connection-factory-class-name="oracle.jdbc.pool.OracleDataSource"
url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=dbhost)(PORT=5521))(CONNECT_DATA=
(SERVICE_NAME=dbhostservice)))"
user="system"
password="manager"
initial-pool-size="4"
min-pool-size="2"
 max-pool-size="10"
shared="true"
  >
<data-source data-source-name="pds1"
 user="system"
 password="manager"
 service="pdb1_service"
 description="pdb1 data source"
  />
<data-source data-source-name="pds2"
user="system"
 password="manager"
service="pdb2_service"
 description="pdb2 data source"
/>
```

```
    </connection-pool>
    </ucp-properties>
```

Monitoring Universal Connection Pool JDBC Resources

Learn about monitoring Universal Connection Pool JDBC Resources using the WebLogic Remote Console or the `JDBCUCPDataSourceRuntimeMBean`, `JDBCDataSourceRuntimeMBean` .

The `JDBCUCPDataSourceRuntimeMBean` 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, and the highest number of active connections. This MBean extends the `JDBCDataSourceRuntimeMBean` so that it can be returned with the list of other JDBC MBeans from the JDBC service. See JDBCUCPDataSourceRuntimeMBean in the *MBean Reference for Oracle WebLogic Server*.

In addition to runtime statistics, the `testPool()` operation returns null if the test is success or an error string otherwise (similar to other data source types). Testing the pool is done only if `SQLForValidateConnection` is set to a SQL string to be executed for validation (for example `SELECT 1 from DUAL`). The rest of the operations will take no action.

To understand more about JDBC monitoring, see Monitoring WebLogic JDBC Resources.

Oracle Sharding Support

Sharding is a data tier architecture in which data is horizontally partitioned across independent databases.

Oracle sharding is available in 12.2 UCP and surfaced from WebLogic Server by using the native UCP data source feature. See Overview of Oracle Sharding in *Using Oracle Sharding*.

After the UCP data source is accessed using a JNDI lookup, the sharding APIs can be used, as seen in the following Java code:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import java.sql.Connection;
import oracle.ucp.jdbc.PoolDataSource;

Context cts = new InitialContext();

/// Look up the data source using JNDI
PoolDatasource pds = (PoolDataSource) ctx.lookup("shardDataSource");

 // Create a key corresponding to sharding key columns, to access the correct 
shard
OracleShardingKey key = pds.createShardingKeyBuilder().subkey(100, 
JDBCType.NUMERIC).build();

 // Fetch a connection to the shard corresponding to the key
Connection conn = pds.createconnectionBuilder().shardingKey(key).build();

 // Use the above connection for performing shard specific operations
```

5 JDBC Data Source Transaction Options

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

- **XA drivers:** For XA drivers, the system automatically selects the **Two-Phase Commit** protocol for global transaction processing.
- **Non-XA drivers:** For non-XA drivers, local transactions are supported by definition. In addition, WebLogic Server offers **Global Transactions** options too.

  You must select **Support Global Transactions** 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 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.

  > **Note:**
  >
  > Logging Last Resource is not supported for data sources used by a multi data source except when used with Oracle RAC version 10g Release 2 (10gR2) and greater versions as described in Administrative Considerations and Limitations for LLR Data Sources.

  – **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.

  – **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
- Understanding the Logging Last Resource Transaction Option

- 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.
- Local Transaction Completion when Closing a Connection

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 JMS services, 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 with 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 more details about the Logging Last Resource transaction processing, see Logging Last Resource Transaction Optimization in *Developing JTA Applications for Oracle WebLogic Server*.

- Advantages to Using the Logging Last Resource Optimization
- Enabling the Logging Last Resource Transaction Optimization
- Programming Considerations and Limitations for LLR Data Sources
- Administrative Considerations and Limitations for LLR Data Sources

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 more information about performance tuning with LLR, see Optimizing Performance with LLR in *Developing JTA Applications for Oracle WebLogic Server*.

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.

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 *Developing JTA Applications for Oracle WebLogic Server*.

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, the connection will be independent, and will not be associated with any subsequently started global transaction. The connection will operate in the `autoCommit(true)` mode. In this mode, every update will commit automatically on its own, and there will be no way to roll back any update unless application code has explicitly set the `autoCommit` state to false and is explicitly managing its own local 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 *Developing JTA Applications for Oracle WebLogic Server*.
- 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.
  - 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. Your database administrator can manually create a table with larger columns if this occurs. See Oracle WebLogic Extensions to JTA in *Developing JTA Applications for Oracle WebLogic Server*.

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 *Developing JTA Applications for Oracle WebLogic Server*.

- 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`.
- 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. See Recovering Transactions For a Failed Clustered Server in *Developing JTA Applications for Oracle WebLogic Server*.
- The LLR transaction option is not permitted for use in JDBC application modules.
- When using Multi Data Sources, the LLR transaction option can only be used with Oracle RAC version 10g Release 2 (10gR2) and greater versions with the following settings:
  - All WebLogic application database JDBC interactions must use the `READ_COMMITTED` transaction isolation level (the default).
  - The Oracle RAC setting `MAX_COMMIT_PROPAGATION_DELAY` must be set to a value of 0 (zero, the default).

  The use of LLR with Multi Data Sources is supported only with Oracle RAC. All (or none) of the members of the Multi Data Source must be LLR data sources.
  - When using Oracle RAC, at least one of the members of the Multi Data Source must be available for recovery processing when the server is booted or the server fails to boot.
  - When not using Oracle RAC, all of the members of the Multi Data Source must be available for recovery processing when the server is booted or the server fails to boot.
- If you use credential mapping or identity pooling 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**.
- When using LLR with a Sybase DBMS, Sybase's JDBC driver requires that certain JDBC stored procedures be installed in the DBMS in order to implement some standard JDBC metadata methods. See the *Sybase jConnect* documentation for details.

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 **Configuration > General** tab of a data source configuration.

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. Oracle 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 *Developing JDBC Applications for Oracle WebLogic Server*.

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

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
- Cannot Recover Pending Transactions
- Possible Performance Loss with Non-XA Resources in Multi-Server Configurations
- Multiple Non-XA Participants

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 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.

Multiple Non-XA Participants

If you use more than one non-XA resource in a global transaction, it is possible to see JTA `SystemExceptions` in the event of a non-atomic outcome. The chance for non-atomic outcomes and `SystemExceptions` tends to increase with the number of two-phase-emulated data source participants.

Note:

The use of a two-phase-emulated data source in a JTA transaction across domains of different versions is not supported.

Local Transaction Completion when Closing a Connection

For a non-XA connection, the `setAutoCommit(true)` method is called if the connection is currently in auto-commit false state when a connection is closed. Per the Jakarta EE JDBC specification, this method automatically commits any outstanding local transaction.
There are some drivers (Oracle 10.x and 11.x driver) that do not commit the local transaction. If the application does not complete (commit or rollback) the local transaction before closing the

connection, a connection is returned to the pool with outstanding work and that work may never be completed or it may be committed or rolled back by the next reservation of that connection. To prevent that situation from happening, a WebLogic data source calls commit on the connection when returning it to the pool, if running with the Oracle 10.x or 11.x driver. If an explicit commit is desired on close, then set the system property `weblogic.datasource.endLocalTxOnNonXaConWithCommit=true` .

Some users may want an abandoned local transaction to rollback instead of commit on close. Setting the following properties will cause local transactions to be rolled back instead of committed if abandoned:

```
-Dweblogic.datasource.endLocalTxOnNonXaConWithCommit=false
-Dweblogic.datasource.endLocalTxOnNonXaConWithRollback=true
```

Note:

It is not a good programming practice to leave abandoned transactions. It is recommended that applications explicitly commit or rollback local transactions.

For an `XA connection`, WebLogic data sources have always rolled back any local transaction when closing the connection. The transaction can be committed instead of rolled back by setting the system property `weblogic.datasource.endLocalTxOnXaConWithCommit=true`.

For an `XA connection`, WebLogic data sources have always rolled back any local transaction when closing the connection. The transaction can be committed instead of rolled back by setting the system property `weblogic.datasource.endLocalTxOnXaConWithCommit=true`.

6 Advanced Configurations for Oracle Drivers and Databases

Oracle provides advanced configuration options such as JDBC Replay Driver (also referred to as Application Continuity Driver), database resident connection policy, global database services to improve data source and driver performance when using Oracle drivers and databases. These configuration options help in management of connection reservation in the data source.

- JDBC Replay Driver
  JDBC Replay Driver (also referred to as Application Continuity Driver) is a general purpose, application-independent infrastructure for Active GridLink and Generic data sources that enables the recovery of work from an application perspective and masks many system, communication, and hardware failures.
- Database Resident Connection Pooling
  Database Resident Connection Pooling (DRCP) provides the ability for multiple web-tier and mid-tier data sources to pool database server processes and sessions that are resident in an Oracle database.
- Global Data Services
  Global Data Services (GDS) enables you to use a global service to provide seamless central management in a distributed database environment. A global server provides automated load balancing, fault tolerance and resource utilization across multiple RAC and single-instance Oracle databases interconnected by replication technologies such as Data Guard or GoldenGate.
- Container Database with Pluggable Databases
  Container Database (CDB) is an Oracle Database feature that minimizes the overhead of having many of databases by consolidating them into a single database with multiple Pluggable Databases (PDB) in a single CDB.
- Service Switching
  Learn about the limitations of service switching.

JDBC Replay Driver

JDBC Replay Driver (also referred to as Application Continuity Driver) is a general purpose, application-independent infrastructure for Active GridLink and Generic data sources that enables the recovery of work from an application perspective and masks many system, communication, and hardware failures.

In today's environment, application developers are required to deal explicitly with outages of the underlying software, hardware, communications, and storage layers. As a result, application development is complex and outages are exposed to the end users. For example, some applications warn users not to hit the submit button twice. When the warning is not heeded, users may unintentionally purchase items twice or submit multiple payments for the same invoice.

JDBC Replay Driver semantics assure that end-user transactions can be executed on time and at-most-once. The only time an end user should see an interruption in service is when the outage is such that there is no point in continuing.

The following topics provide information on how to configure and use JDBC Replay Driver:

> **Note:**
>
> WLS Multi Data Source does not support Transparent Application Continuity (TAC).

- How JDBC Replay Driver Works
- Requirements and Considerations
- Configuring JDBC Replay Driver
- Viewing Runtime Statistics for JDBC Replay Driver
- JDBC Replay Driver Auditing
- Limitations with JDBC Replay Driver with Oracle 12c Database

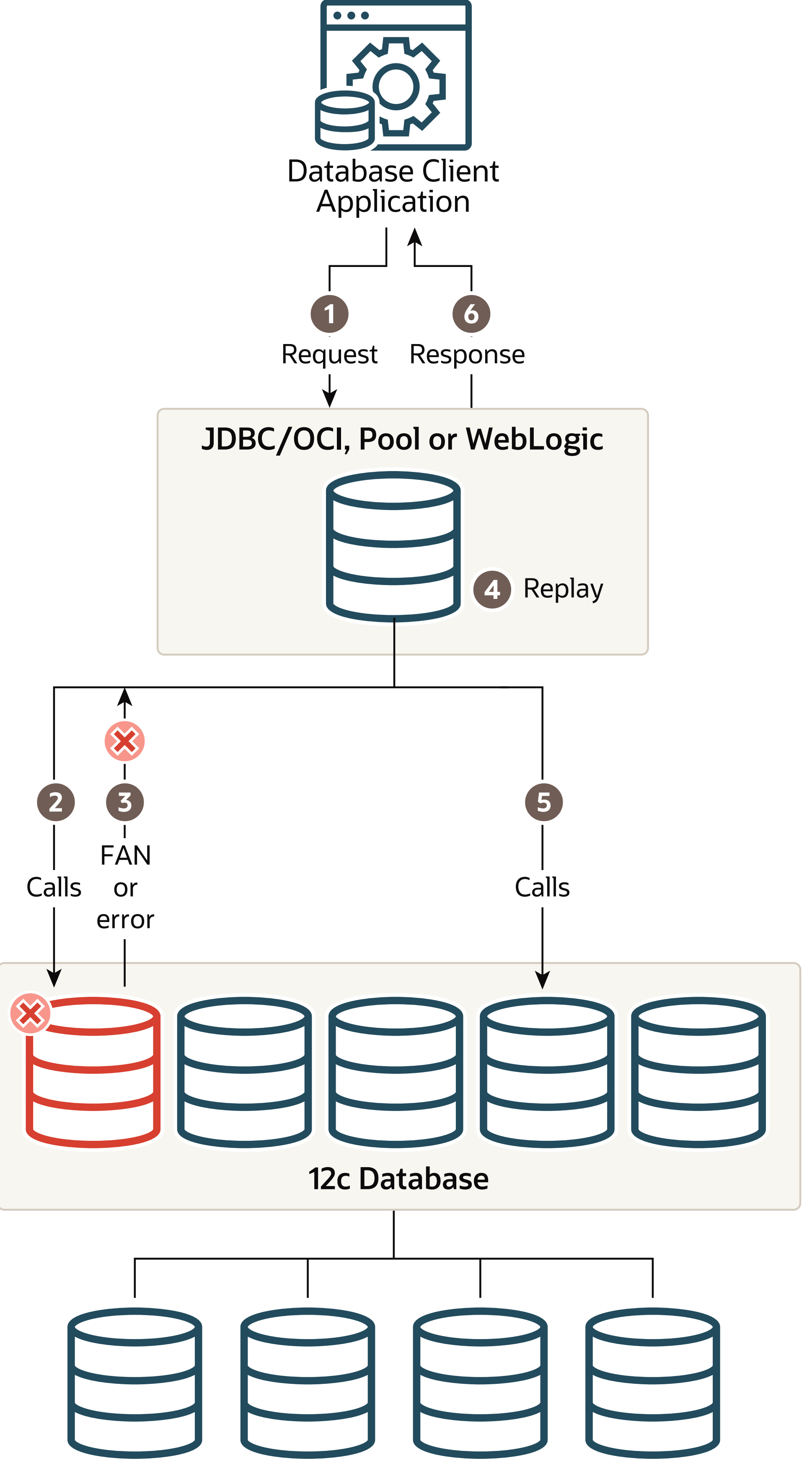
How JDBC Replay Driver Works

Following any outage that is due to a loss of database service, planned or unplanned, JDBC Replay Driver rebuilds the database session. Once an outage is identified by Fast Application Notification or a recoverable `ORACLE` error, the Oracle driver:

- Establishes a new database session to clear any residual state.
- If a callback is registered, issues a callback allowing the application to re-establish initial state for that session.
- Executes the saved history accumulated during the request.

The Oracle driver determines the timing of replay calls. Calls may be processed chronologically or using a lazy processing implementation depending on how the application changes the database state. The replay is controlled by the Oracle 12c Database Server. For a replay to be approved, each replayed call must return exactly the same client visible state that was seen and potentially used by the application during the original call execution.

Figure 6-1 JDBC Replay Driver



Requirements and Considerations

The following section provides requirements and items to consider when using JDBC Replay Driver (also referred as Application Continuity) with WebLogic applications:

- Requires an Oracle 12c JDBC driver and database. See Using an Oracle 12c Database.
- JDBC Replay Driver supports read and read/write transactions. XA transactions are not supported. To support transactions using non-XA drivers such as an Oracle driver for JDBC Replay Driver, see Enabling Support for Global Transactions with a Non-XA JDBC Driver in *Administering JDBC Data Sources for Oracle WebLogic Server* for information.

> **Note:**
>
> Remember to call `connection.setAutoCommit(false)` in your application to prevent breaking the transaction semantics and disabling JDBC Replay Driver in your environment.

- Deprecated `oracle.sql.*` concrete classes are not supported. Occurrences should be changed to use either the corresponding `oracle.jdbc.*` interfaces or `java.sql.*` interfaces. Oracle recommends using the standard `java.sql.*` interfaces. See Using API Extensions for Oracle JDBC Types in *Developing JDBC Applications for Oracle WebLogic Server*.
- JDBC Replay Driver works by storing intermediate results in memory. An application may run slower and require significantly more memory than running without the feature.
- If the WebLogic statement cache is configured with JDBC Replay Driver, the cache is cleared every time the connection is replayed.
- There are additional limitations and exceptions to the JDBC Replay Driver feature which may affect whether your application can use Replay. For more information, see Application Continuity for Java in the *Oracle Database JDBC Developer's Guide*.
- The database service that is specified in the URL for the data source must be configured with the failover type set to `TRANSACTION` and the `-commit_outcome` parameter to `TRUE`. For example:

```
srvctl modify service -d mydb -s myservice -e TRANSACTION -commit_outcome TRUE
-rlbgoal SERVICE_TIME -clbgoal SHORT
```

Configuring JDBC Replay Driver

The topic provides information on how to implement JDBC Replay Driver in your environment.

- Selecting the Driver for JDBC Replay Driver
- Using a Connection Callback
- Setting the Replay Timeout
- Disabling JDBC Replay Driver for a Connection
- Configuring Logging for JDBC Replay Driver

Selecting the Driver for JDBC Replay Driver

Configure your data source to use the correct JDBC driver using one of the following methods:

- If you are creating a new data source, when asked to select a Database driver from the drop-down menu in the configuration wizard, select the appropriate Oracle driver that supports JDBC Replay Driver for your environment.
- When creating or editing a data source with a text editor or WLST, set the JDBC driver to `oracle.jdbc.replay.OracleDataSourceImpl`.

See Requirements and Considerations.

Using a Connection Callback

- Create an Initialization Callback
- Registering an Initialization Callback
- Unregister an Initialization Callback

Create an Initialization Callback

To create a connection initialization callback, your application must implement the `initialize(java.sql.Connection connection)` method of the `oracle.ucp.jdbc.ConnectionInitializationCallback` interface. Only one callback can be created per connection pool.

The callback is ignored if a labeling callback is registered for the connection pool. Otherwise, the callback is executed at every connection check out from the pool and at each successful reconnect following a recoverable error at replay. Use the same callback at runtime and at replay to ensure that exactly the same initialization that was used when the original session was established is used during the replay. If the callback invocation fails, replay is disabled on that connection.

Note:

Connection Initialization Callback is not supported for clients (JDBC over RMI).

Connection callback once registered will be called even without Oracle driver.

The following example demonstrates a simple initialization callback implementation:

```
. . .
import oracle.ucp.jdbc.ConnectionInitializationCallback ;
. . .
class MyConnectionInitializationCallback implements ConnectionInitializationCallback {
  public MyConnectionInitializationCallback()  {
  }
  public void initialize(java.sql.Connection connection)  throws SQLException {
     // Re-set the state for the connection, if necessary
  }
}
```

Registering an Initialization Callback

The WLDataSource interface provides the registerConnectionInitializationCallback(ConnectionInitializationCallback callback) method for registering initialization callbacks. Only one callback may be registered on a connection pool. The following example demonstrates registering an initialization callback that is implemented in the MyConnectionInitializationCallback class:

```
. . .
import weblogic.jdbc.extensions.WLDataSource;
. . .
MyConnectionInitializationCallback callback = new MyConnectionInitializationCallback();
((WLDataSource)ds).registerConnectionInitializationCallback(callback);
. . .
```

Unregister an Initialization Callback

The WLDataSource interface provides the unregisterConnectionInitializationCallback() method for unregistering a ConnectionInitializationCallback. The following example demonstrates removing an initialization callback:

```
. . .
import weblogic.jdbc.extensions.WLDataSource;
((WLDataSource)ds).unregisterConnectionInitializationCallback();
. . .
```

Setting the Replay Timeout

Use the ReplayInitiationTimeout attribute on the **Oracle** tab for a data source in the WebLogic Remote Console to set the amount of time a data source allows for JDBC Replay Driver processing before timing out and ending a replay session context for a connection.

For applications that use the WebLogic HTTP request timeout, make sure to set the ReplayInitiationTimeout appropriately:

- You should set the ReplayInitiationTimeout value equal to the HTTP session timeout value to ensure the entire HTTP session is covered by a replay session. The default ReplayInitiationTimeout and the default HTTP session timeout are both 3600 seconds.
- If the HTTP timeout value is longer than ReplayInitiationTimeout value, replay events will not be available for the entire HTTP session.
- If the HTTP timeout value is shorter than the ReplayInitiationTimeout value, your application should close the connection to end the replay session.

Disabling JDBC Replay Driver for a Connection

You can disable JDBC Replay Driver on a per-connection basis using the following:

```
. . .
if (connection instanceof oracle.jdbc.replay.ReplayableConnection) {
 ((oracle.jdbc.replay.ReplayableConnection)connection).disableReplay();
}
. . .
```

You can disable JDBC Replay Driver at the database service level by modifying the service to have a failover type of NONE. For example:

```
srvctl modify service -d mydb -s myservice -e NONE
```

You can also disable JDBC Replay Driver at the data source level by setting the ReplayINitializationTimeout to 0. When set to zero (0) seconds, replay processing (failover) is disabled (begin and endRequest are still called).

Configuring Logging for JDBC Replay Driver

To enable logging of JDBC Replay Driver processing, use the following WebLogic property:

```
-Dweblogic.debug.DebugJDBCReplay=true
```

Use `-Djava.util.logging.config.file=`*`configfile`*, where *`configfile`* is the path and file name of the configuration file property used by standard JDK logging, to control the log output format and logging level.

See *Adding WebLogic Logging Services to Applications Deployed on Oracle WebLogic Server*.

Viewing Runtime Statistics for JDBC Replay Driver

JDBC Replay Driver statistics are available using the `JDBCReplayStatisticsRuntimeMBean` for Active GridLink and Generic data sources.

The `JDBCReplayStatisticsRuntimeMBean`:

- Is available for Active GridLink and Generic data sources. It is not available (null is returned) for Universal Connection Pool and Multi Data Sources.
- Is available only if running with the 12.1.0.2 or later Oracle thin driver. It is not available (null is returned) for earlier driver versions.
- Is available only if the data source is configured to use the JDBC Replay Driver. It is not available (null is returned) for non-replay drivers.
- Will not have any statistics set initially (they will be initialized to -1). You must call the `refreshStatistics()` operation on the MBean to update the statistics before getting them.

Note:

Refreshing the statistics is a heavy operation. It locks the entire pool and runs through all reserved and unreserved connections aggregating the statistics. Running this operation frequently will impact the performance of the data source. Performance can also be impacted when clearing the statistics.

Table 6-1 lists the statistics that you can access using the `JDBCReplayStatisticsRuntimeMBean`.

Table 6-1 Runtime Statistics for JDBCReplayStatisticsRuntimeMBean

Name	Description
`TotalRequests`	Total number of successfully submitted requests.

Table 6-1 (Cont.) Runtime Statistics for JDBCReplayStatisticsRuntimeMBean

Name	Description
`TotalCompletedRequests`	Total number of completed requests.
`TotalCalls`	Total number of JDBC calls executed.
`TotalProtectedCalls`	Total number of JDBC calls executed that are protected by JDBC Replay Driver.
`TotalCallsAffectedByOutages`	Total number of JDBC calls affected by outages. This includes both local calls and calls that involve roundtrip(s) to the database server.
`TotalCallsTriggeringReplay`	Total number of JDBC calls that triggered replay. Replay can be disabled for some requests, therefore not all calls affected by an outage trigger replay.
`TotalCallsAffectedByOutagesDuringReplay`	Total number of JDBC calls affected by outages in the middle of replay. Outages may be cascaded and strike a call multiple times when replay is ongoing. JDBC Replay Driver automatically reattempts replay when this happens, unless it reaches the maximum retry limit.
`SuccessfulReplayCount`	Total number of replays that succeeded. Successful replays mask the outages from applications.
`FailedReplayCount`	Total number of replays that failed. When replay fails, it re-throws the original `SQLRecoverableException` to the application, along with the reason for the failure chained to the original exception. The application can call `getNextException` to retrieve the reason.
`ReplayDisablingCount`	Total number of times that replay is disabled. When replay is disabled in the middle of a request, the remaining calls in that request are no longer protected by JDBC Replay Driver. If an outage strikes one of the remaining calls, no replay is attempted, and the application gets an `SQLRecoverableException`.
`TotalReplayAttempts`	Total number of replay attempts. JDBC Replay Driver automatically reattempts when replay fails, so this number may exceed the number of JDBC calls that triggered replay.

For more information, see JDBCReplayStatisticsRuntimeMBean in *MBean Reference for Oracle WebLogic Server* and ReplayableConnection.StatisticsReportType in *Oracle JDBC Java API Reference*.

Example 6-1 WLST Sample

You can access the statistics on the runtime MBean using WLST. The following sample WLST script shows how to print the information on the `JDBCReplayStatisticsRuntimeMBean`:

```
import sys, socket, os
hostname = socket.gethostname()
datasource='JDBC GridLink Data Source-0'
svr='myserver'
```

```
connect("weblogic","password","t3://"+hostname+":7001")
serverRuntime()
cd('/JDBCServiceRuntime/' + svr + '/JDBCDataSourceRuntimeMBeans/' +
   datasource + '/JDBCReplayStatisticsRuntimeMBean/' +
   datasource + '.ReplayStatistics')
cmo.refreshStatistics()
ls()
total=cmo.getTotalRequests()
cmo.clearStatistics()
```

Example 6-2   Java Sample

The following Java example demonstrates how to expose the statistics using the `JDBCReplayStatisticsRuntimeMBean`:

```
import javax.naming.NamingException;
import javax.management.AttributeNotFoundException;
import javax.management.MBeanServer;
import javax.management.InstanceNotFoundException;
import javax.management.ReflectionException
 import javax.management.ObjectName;
import javax.management.MalformedObjectNameException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanException;
import javax.management.MBeanParameterInfo;
import weblogic.management.runtime.JDBCReplayStatisticsRuntimeMBean;
  public void printReplayStats(String dsName) throws Exception {
     MBeanServer server = getMBeanServer();
     ObjectName[] dsRTs = getJdbcDataSourceRuntimeMBeans(server);
     for (ObjectName dsRT : dsRTs) {
       String name = (String) server.getAttribute(dsRT, "Name");
       if (name.equals(dsName)) {
         ObjectName mb = (ObjectName)server.getAttribute(dsRT,
           "JDBCReplayStatisticsRuntimeMBean");
         server.invoke(mb, "refreshStatistics", null, null);
        MBeanAttributeInfo[] attributes =
           server.getMBeanInfo(mb).getAttributes();
         for (int i = 0; i < attributes.length; i++) {
           if (attributes[i].getType().equals("java.lang.Long")) {
             System.out.println(attributes[i].getName()+"="+
               (Long) server.getAttribute(mb, attributes[i].getName()));
           }
         }
       }
     }
   }
  MBeanServer getMBeanServer() throws Exception {
     InitialContext ctx = new InitialContext();
     MBeanServer server = (MBeanServer) ctx.lookup("java:comp/env/jmx/
runtime");
     return server;
   }
  ObjectName[] getJdbcDataSourceRuntimeMBeans(MBeanServer server)
       throws Exception {
     ObjectName service = new ObjectName(
```

```
"com.bea:Name=RuntimeService,Type=weblogic.management.mbeanservers.runtime .Ru
ntimeServiceMBean");
     ObjectName serverRT = (ObjectName) server.getAttribute(service,
       "ServerRuntime");
     ObjectName jdbcRT = (ObjectName) server.getAttribute(serverRT,
       "JDBCServiceRuntime");
     ObjectName[] dsRTs = (ObjectName[]) server.getAttribute(jdbcRT,
       "JDBCDataSourceRuntimeMBeans");
     return dsRTs;
   }
```

JDBC Replay Driver Auditing

During a `ConnectionInitializationCallback`, between the first connection initialization and reinitialization during replay, the application may want to know when the connection work is being replayed. The `getReplayAttemptCount()` method on the WLConnection interface is available to get the number of times that replay is attempted on the connection.

When a connection is first being initialized, it will be set to 0. For subsequent initialization of the connection when it is being replayed, it will be set to a value greater than 0.

Note:

This counter only indicates attempted replays since it is possible for replay to fail for various reasons (after which the connection is no longer valid). For a non-replay driver, it will always return 0.

Example 6-3 WLST Sample

The following is a sample callback class for initializing the connection. It assumes that there is some mechanism for getting an application identifier associated with the current work or transaction.

```
import java.sql.SQLException;
import java.util.Date;
import java.text.SimpleDateFormat;
import java.util.Properties;
import weblogic.jdbc.extensions.WLConnection;
import oracle.ucp.jdbc.ConnectionInitializationCallback;
public class callback implements ConnectionInitializationCallback {
final String idLabel = "GUUID";
public callback() {
}
public void initialize(java.sql.Connection conn) throws SQLException {
if (((WLConnection)conn).getReplayAttemptCount() == 0) {
// first time - initialize the label value
((WLConnection)conn).applyConnectionLabel(idLabel, Application.getGuuid());
// Get the id from somewhere and store it in the connection label
} else {
Properties props = ((WLConnection)conn).getConnectionLabels();
String value = props.getProperty(idLabel);
System.out.println("Transaction '"+value+"' is getting replayed at " + new
```

```
SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new
Date()));
}
}
}
```

Limitations with JDBC Replay Driver with Oracle 12c Database

The following section provides information on limitations when using Oracle Database Release 12c with JDBC Replay Driver:

- Database Resident Connection Pooling is not supported. A web request is not replayed and the `original java.sql.SQLRecoverableException` is thrown if an outage occurs.
- Cannot be used with PDB tenant switching using `ALTER SESSION SET CONTAINER`.

Database Resident Connection Pooling

Database Resident Connection Pooling (DRCP) provides the ability for multiple web-tier and mid-tier data sources to pool database server processes and sessions that are resident in an Oracle database.

See Database Resident Connection Pooling in *JDBC Developer's Guide*.

- Requirements and Considerations
- Configuring DRCP

Requirements and Considerations

The following section provides requirements and items to consider when using DRCP with WebLogic applications:

- Requires an Oracle 12c JDBC Driver and Database. See Using an Oracle 12c Database.
- If the WebLogic statement cache is configured along with DRCP, the cache is cleared every time the connection is returned to the pool with `close()`.
- WebLogic Server data sources do not support connection labeling on DRCP connections and a `SQLException` is thrown. For example, using `getConnection` with properties on `WLDataSource` or a method on `LabelableConnection` is called, generates an exception. Note, `registerConnectionLabelingCallback` and `removeConnectionLabelingCallback` on `WLDataSource` are permitted.
- WebLogic Server does not support defining the `oracle.jdbc.DRCPConnectionClass` as a system property. It must be defined as a connection property in the data source descriptor.
- For DRCP to be effective, applications must return connections to the connection pool as soon as work is completed. Holding on to connections and using harvesting defeats the use of DRCP.
- When using DRCP, the JDBC program must attach to the server before performing operations on the connection and must detach from the server to allow other connections to use the pooled session. By default, when the JDBC program is attaching to the server, it does not actually reserve a session but returns and defers the reservation until the next database round-trip. As a result, the subsequent database operation may fail because it cannot reserve a session. To prevent this from happening, there is a network timeout value that forces a round-trip to the database after an attach to the server. Once this occurs, the

network timer is unset. The default network timeout is 10,000 milliseconds. You can set it to another value by setting the system property `weblogic.jdbc.attachNetworkTimeout`.

This property is the timeout, in milliseconds to wait for the attach to be done and the database round trip to return. If set to 0, then the additional processing around the server attach is not done.

For more information on configuring DRCP, see Configuring Database Resident Connection Pooling in the *Oracle® Database Administrator's Guide*.

Configuring DRCP

You must configure datasource and database for DRCP in your environment.

- Configuring a Data Source for DRCP
- Configuring a Database for DRCP

Configuring a Data Source for DRCP

To configure your data source to support DRCP:

- If you are creating a new data source, on the **Connection Properties** tab of the data source configuration wizard, under **Additional Configuration Properties**, enter the DRCP connection class in the `oracle.jdbc.DRCPConnectionClass` field.

  In the resulting data source:

  – The suffix `:POOLED` is added to the constructed short-form of the URL. For example: `jdbc:oracle:thin:@//host:port/service_name:POOLED`

  – For the service form of the URL, `(SERVER=POOLED)` is added after the `(SERVICE_NAME=name)` parameter in the `CONNECT_DATA` element.

  – The value/name pair of the DRCP connection class appears as a connection property on the **Connection Pool** tab. For example: `oracle.jdbc.DRCPConnectionClass=myDRCPclass`.

- When creating or editing a data source with a text editor or using WLST:

  1. Change the `URL` element to include a suffix of `:POOLED` or `(SERVER=POOLED)` for service URLs. For example: `<url>jdbc:oracle:thin:@host:port:service:POOLED</url>`.
  2. Update the connection properties to include the value/name pair of the DRCP connection class. For example:

```
<properties>
      <property>
        <name>aname</name>
        <value>avalue</value>
      </property>
      <property>
        <name>oracle.jdbc.DRCPConnectionClass</name>
        <value>myDRCPclass</value>
      </property>
</properties>
```

- WebLogic Server throws a configuration error if the `DataSource` resource definition has a `oracle.jdbc.DRCPConnectionClass` connection property or a `POOLED` URL but not both.

This validation is performed when testing the connection listener in the console, deploying a `DataSource` during system boot, or when targeting a `DataSource`.

- Set `TestConnectionsOnReserve=true` to minimize problems with `MAX_THINK_TIME`. See Configuring a Database for DRCP.
- Set `TestFrequencySeconds` to a value less than `INACTIVITY_TIMEOUT`. See Configuring a Database for DRCP.

Configuring a Database for DRCP

To configure your Oracle database to support DRCP:

- DRCP must be enabled on the database side using:

  ```
  SQL> DBMS_CONNECTION_POOL.CONFIGURE_POOL('SYS_DEFAULT_CONNECTION_POOL')
  ```

  ```
  SQL> EXECUTE DBMS_CONNECTION_POOL.START_POOL();
  ```

- The following parameters of the server pool configuration must be set correctly:
  - `MAXSIZE`: The maximum number of pooled servers in the pool. The default value is 40. The connection pool reserves 5% of the pooled servers for authentication and at least one pooled server is always reserved for authentication. When setting this parameter, ensure that there are enough pooled servers for both authentication and connections.

    It may be necessary to set `MAXSIZE` to the size of the largest WebLogic connection pool using the DRCP.
  - `INACTIVITY_TIMEOUT`: The maximum time, in seconds, the pooled server can stay idle in the pool. After this time, the server is terminated. The default value is 300. This parameter does not apply if the server pool has a size of `MINSIZE`.

    If a connection is reserved from the WebLogic data source and then not used, the inactivity timeout may occur and the DRCP connection will be released. Set `INACTIVITY_TIMEOUT` appropriately or return connections to the WebLogic data source if they will not be used. You can also use `TestFrequencySeconds` to ensure that unused connections don't time out.
  - `MAX_THINK_TIME`: The maximum time of inactivity, in seconds, for a client after it obtains a pooled server from the pool. After obtaining a pooled server from the pool, if the client application does not issue a database call for the time specified by `MAX_THINK_TIME`, the pooled server is freed and the client connection is terminated. The default value is 120.

    If a connection is reserved from the WebLogic data source and no activity is done within the `MAX_THINK_TIME`, the connection is released. You can set `Test Connections On Reserve` (see Testing Reserved Connections) or set `MAX_THINK_TIME` appropriately. You can minimize the overhead of testing connections by setting `SecondstoTrustanIdlePoolConnection` to a reasonable value less than `MAX_THINK_TIME`. See Tuning Data Source Connection Pools.

  If the server pool configuration parameters are not set correctly for your environment, your data source connections may be terminated and your applications may receive an error, such as a socket exception, when accessing a WebLogic data source connection.

Global Data Services

Global Data Services (GDS) enables you to use a global service to provide seamless central management in a distributed database environment. A global server provides automated load

balancing, fault tolerance and resource utilization across multiple RAC and single-instance Oracle databases interconnected by replication technologies such as Data Guard or GoldenGate.

- Requirements and Considerations
- Creating a Active GridLink Data Source for GDS Connectivity

Requirements and Considerations

Ensure to complete the following requirements and considerations when using Global Data Services in WebLogic Server:

- Requires an Oracle 12c JDBC driver and database. See Using an Oracle 12c Database.
- It is not possible to use a single SCAN address to replace multiple Global Service Manager (GSM) addresses.
- For update operations to be handled correctly, you must define a service for updates that is only enabled on the primary database.
- Define a separate service for read-only operations that is located on the primary and secondary databases.
- Since only a single service can be defined for a URL and a single URL for a data source configuration, one data source must be defined for the update service and another data source defined for the read-only service.
- Your application must be written so that update operations are process by the update data source and read-only operations are processed by the read-only data source.

Creating a Active GridLink Data Source for GDS Connectivity

Use the WebLogic Remote Console to create a Active GridLink data source that uses a modified URL to provide Global Data Services (GDS) connectivity.

The connection information for a GDS URL is similar to a RAC Cluster, containing the following basic information:

- Service name (Global Service Name)
- Address/port pairs for Global Service Manager
- GDS Region in the `CONNECT_DATA` parameter

The following is a sample URL:

```
jdbc:oracle:thin:@(DESCRIPTION=
    (ADDRESS_LIST=(LOAD_BALANCE=ON)(FAILOVER=ON)
         (ADDRESS=(HOST=myHost1.com)(PORT=1111)(PROTOCOL=tcp))
         (ADDRESS=(HOST=myHost2.com)(PORT=2222)(PROTOCOL=tcp)))
    (CONNECT_DATA=(SERVICE_NAME=my.gds.cloud)(REGION=west)))
```

Container Database with Pluggable Databases

Container Database (CDB) is an Oracle Database feature that minimizes the overhead of having many of databases by consolidating them into a single database with multiple Pluggable Databases (PDB) in a single CDB.

See Managing Pluggable Databases in *Enterprise Manager Lifecycle Management Administrator's Guide*.

- Creating Service for PDB Access
- DRCP and CDB/PDB
- Setting the PDB using JDBC

Creating Service for PDB Access

Access to a PDB is completely transparent to a WebLogic Server data source. It is accessed like any other database using a URL with a service. The service must be associated with the PDB. It can be created in SQLPlus by associating a session with the PDB, creating the service and starting it.

```
alter session set container = cdb1_pdb1; -- configure service for each PDB
execute
dbms_service.create_service('replaytest_cdb1_pdb1.regress.rdbms.dev.us.myCompany.com','re
playtest_cdb1_pdb1.regress.rdbms.dev.us.myCompany.com');
execute
DBMS_SERVICE.START_SERVICE('replaytest_cdb1_pdb1.regress.rdbms.dev.us.myCompany.com');
```

If you want to set up the service for use with JDBC Replay Driver, it needs to be appropriately configured. For example, SQLPlus:

```
declare
params dbms_service.svc_parameter_array ;
begin
params('goal') := 'service_time' ;
params('commit_outcome') := 'true' ;
params('aq_ha_notifications') := 'true' ;
params('failover_method') := 'BASIC' ;
params('failover_type') := 'TRANSACTION' ;
params('failover_retries') := 60 ;
params('failover_delay') := 2 ;
dbms_service.modify_service('replaytest_cdb1_pdb1.regress.rdbms.dev.us.myCompany.com',
params);
end;
/
```

DRCP and CDB/PDB

DRCP cannot be used in a PDB. It must be associated with a CDB only. To configure, set a session to point to the CDB and start the DRCP pool. For example:

```
alter session set container = cdb$root;
execute dbms_connection_pool.configure_pool('SYS_DEFAULT_CONNECTION_POOL');
execute dbms_connection_pool.start_pool();
```

Setting the PDB using JDBC

Initially when a connection is created for the pool, it is created using the URL with the service associated with a specific PDB in a CDB. It is possible to dynamically change the PDB within the same CDB. Changing PDB's is done by executing the SQL statement:

```
ALTER SESSION SET CONTAINER = name SET SERVICE servicename;
```

Specifying `SET SERVICE servicename` allows for an explicit service to be configured by the application and named. This allows for support of Load Balancing Advisories, Session Affinity, Fast Application Notification, JDBC Replay Driver, and Proxy Authentication. These features are not available without the *SET SERVICE servicename* clause.

After the container is changed, the following do not change:

- The RAC instance
- The connection object
- The WebLogic connection lifecycle (`enabled`/`disabled`/`destroyed`)
- The WebLogic connection attributes.

Any remaining state on the connection is cleared to avoid leaking information between PDB's.

If configured, the following are reset:

- JDBC Replay Driver
- DRCP
- `client identifier`
- proxy user
- The connection harvesting callback

> **Note:**
>
> DRCP is not supported with PDB switching

Service Switching

Learn about the limitations of service switching.

The limitations of using service switching with WebLogic Server are as follows:

- Service switching has no impact on where the service is offered.
- Service switching is only allowed only when the service is published at that instance.
- Service switching is only allowed at request boundaries. This is necessary for JDBC Replay Driver to work correctly.
- Service switching is only allowed at a top level call (no user call is active).
- Service switching is not supported with Database Resident Connection Pooling.
- Service switching returns an error if there is an open transaction, local, or XA.
- Service attributes set at switch are never carried over from earlier usage. The application must set up the session appropriately.
- Service switching is supported in non-CDB environments as wells as CDB environments. In the non-CDB environment, the container cannot change.
- As with the earlier version, the service name may change during the switch but the instance name may not change.
- XA affinity is based on a `service_name`, `database_name`, `instance_name triple`. When the service changes, there is no XA affinity enforced.

Note:

There is a limitation for Generic, Active GridLink, and Universal Connection Pool data sources. Fast Application Notification and Fast Connection Failover (FCF) are service based. When the data source is created, a subscription is set up for the configured service name. The data source will receive events for instance and service up and down. When the application switches the service, service up and down events will not be received for the new service name. Since gradual draining and scheduled maintenance are based on stopping the service allowing connections to drain before the instance is stopped, scheduled maintenance (planned down) does not work with application service switching. When the instance is stopped, a down event will be processed and the connections closed. WebLogic Server shared pooling manages multiple subscriptions and the resulting Fast Application Notification service events properly.

7 Using Connection Harvesting

Connection harvesting helps to ensure that a specified number of connections are always available in the pool and improves performance by minimizing connection initialization. Using connection harvesting in your applications involves enabling the use of connection harvesting, marking the connection pools as harvestable, and recovering the harvested connections.

- What is Connection Harvesting
  Connection harvesting is particularly useful if an application caches connection handles. Caching is typically performed for performance reasons because it minimizes the initialization of state necessary for connections to participate in a transaction.
- Enable Connection Harvesting
  To enable and specify a threshold to trigger connection harvesting, use Connection Harvest Trigger Count data source attribute.
- Marking Connections Harvestable
  When connection harvesting is enabled, all connections are initially marked harvestable.
- Recover Harvested Connections
  When a connection is harvested, an application callback is executed to cleanup the connection if the callback has been registered. A unique callback must be generated for each connection; generally it needs to be initialized with the connection object.

What is Connection Harvesting

Connection harvesting is particularly useful if an application caches connection handles. Caching is typically performed for performance reasons because it minimizes the initialization of state necessary for connections to participate in a transaction.

For example: A connection is reserved from the data source, initialized with necessary session state, and then held in a context object. Holding connections in this manner may cause the connection pool to run out of available connections. Connection harvesting appropriately reclaims the reserved connections and allows the connections to be reused.

> **Note:**
>
> In WebLogic Server 12.2.1.1.0 and earlier releases, do not enable harvesting on data sources that are referenced by WebLogic JDBC or TLOG-in-DB stores. Harvesting may destabilize such stores, which can in turn destabilize WebLogic JMS or WebLogic JTA. If you want to enable harvesting on a data source used by WebLogic JDBC or TLOG-in-DB stores in WebLogic Server 12.2.1.1.0 and earlier releases, contact Oracle Support for a patch that protects their store connections from getting harvested.

For more information, see Using Connection Harvesting.

Enable Connection Harvesting

To enable and specify a threshold to trigger connection harvesting, use Connection Harvest Trigger Count data source attribute.

For example, if `Connection Harvest Trigger Count` is set to 10, connection harvesting is enabled and the data source begins to harvest reserved connections when the number of available connections drops to 10. A value of -1, the default, indicates that connection harvesting is disabled.

When connection harvesting is triggered, the *`Connection Harvest Max Count`* specifies how many reserved connections should be returned to the pool. The number of connections actually harvested ranges from 1 to the value of `Connection Harvest Max Count`, depending on how many connections are marked harvestable.

Marking Connections Harvestable

When connection harvesting is enabled, all connections are initially marked harvestable.

If you do not want a connection to be harvestable, you must explicitly mark it as unharvestable by calling the `setConnectionHarvestable(`*`boolean`*`)` method in the `oracle.ucp.jdbc.HarvestableConnection` interface with `false` as the argument value. For example, use the following statements to prevent harvesting when a transaction is used within a transaction:

```
. . .
Connection conn = datasource.getConnection();
((HarvestableConnection) conn).setConnectionHarvestable(false);
. . .
```

After work with the connection is completed, you can mark the connection as harvestable by setting `setConnectionHarvestable(true)` so the connection can be harvested if required. You can tell the harvestable status of a connection by calling `isConnectionHarvestable()`.

Recover Harvested Connections

When a connection is harvested, an application callback is executed to cleanup the connection if the callback has been registered. A unique callback must be generated for each connection; generally it needs to be initialized with the connection object.

For example:

```
import java.sql.Connection;
 . . .
public myHarvestingCallback implements ConnectionHarvestingCallback {
  private Connection conn;
  mycallback(Connection conn) {
    this.conn = conn;
  }
  public boolean cleanup() {
    try {
       conn.close();
    } catch (Exception ignore) {
       return false;
    }
    return true;
```

```
  }
}
. . .
Connection conn = ds.getConnection();
try {
  (HarvestableConnection)conn).registerConnectionHarvestingCallback(
    new myHarvestingCallback(conn));
  (HarvestableConnection)conn).setConnectionHarvestable(true);
} catch (Exception exception) {
  // This can't be from registration - setConnectionHarvestable must have failed.
  // That most likely means that the connection has already been harvested.
  // Do whatever logic is necessary to clean up here and start over.
   throw new Exception("Need to get a new connection");
}
. . .
```

Note:

Consider the following:

- After a connection is harvested, an application can only call `Connection.close`.
- If the connection is not closed by the application, a warning is logged indicating that the connection was forced closed if LEAK profiling is enabled.
- If the callback throws an exception, a message is logged and the exception is ignored. Use `JDBCCONN` debugging to retrieve a full stack trace.
- The return value of the cleanup method is ignored.
- Connection harvesting releases reserved connections that are marked harvestable by the application when a data source falls to a specified number of available connections. By default, this check is performed every 30 seconds. You can tune this behavior using the `weblogic.jdbc.harvestingFrequencySeconds` system property which specifies the amount of time, in seconds, the system waits before harvesting marked connections. Setting this system property to less than 1 disables harvesting.

8 Using Shared Pooling Data Sources

> **Note:**
>
> Shared Pooling Data Sources is deprecated in WebLogic Server as of version 14c (14.1.2.0.0).

Shared pooling provides the ability for Multi Data Source definitions to share an underlying connection pool. This feature improves connection utilization and density when data sources are not heavily used by applications, or are not participating in long running transaction processing. When configured to connect to an Oracle Container Database (CDB) environment, the shared pool can easily manage connections to multiple Pluggable Database (PDB) services.

- How Shared Pooling Works
- Requirements and Considerations when using Shared Pooling Data Sources
  Learn and understand the requirements for using shared pooling.
- Configuring Shared Pooling
  You can configure shared pooling by setting WebLogic Server-specific driver properties and configuring Database properties.

How Shared Pooling Works

When an application component requests a connection from a data source, the shared pool attempts to locate a connection that matches the database name and service for the data source.If an existing connection is found, it is returned to the application. Otherwise, an available connection associated with a different database is reserved and re-purposed for the requesting data source.

> **Note:**
>
> If there are no available connections in the shared pool, and if capacity is available, a new connection will be created for the common service and repurposed for the shared data source requesting the connection.

Requirements and Considerations when using Shared Pooling Data Sources

Learn and understand the requirements for using shared pooling.

- All sharing data source definitions that specify a particular shared pool JNDI name must have compatible configuration attributes with the shared pooling data source definitions.

- Separate ONS subscriptions need to be managed for each sharing data source that defines a unique PDB service.
- The shared pool feature provides support for Generic and Active GridLink (AGL) data source types.

> **Note:**
>
> A Generic sharing data source cannot be used as a member data source in a Multi Data Source configuration, as this would result in an exception being raised at runtime causing the Multi Data Source deployment to fail.

- The shared connection pool supports connection matching based on PDB name, service name and RAC instance name (AGL).
- Shared pooling does not provide support for Pinned-to-thread Optimization, Application Invoked PDB Switch, and Identity-based pooling and BI Impersonation identity options.

Configuring Shared Pooling

You can configure shared pooling by setting WebLogic Server-specific driver properties and configuring Database properties.

- Configuring WebLogic Server-Specific Driver Properties for Shared Pooling
- Configuring Database for Shared Pooling

Configuring WebLogic Server-Specific Driver Properties for Shared Pooling

To configure shared pooling in your environment you need to set the following properties in the data source:

Setting the Shared Pool Definition

- Set the shared pool attribute `weblogic.jdbc.sharedPool=true`.

> **Note:**
>
> This attribute indicates whether the data source definition represents a shared connection pool. When this attribute is set to *false* (default) any data source referencing the data source as a shared pool will result in a deployment exception.

Setting the Sharing Data Source Definition

- Set the shared pool JNDI Name.

```
weblogic.jdbc.sharedPoolJNDIName=<jndiname>
```

> **Note:**
>
> If the data source bound under JNDI Name is not configured as a shared pool then it will result in deployment exception.

- Set the name of the PDB that is associated with the sharing data source.

  ```
  weblogic.jdbc.pdbName=<pdb>
  ```

  > **Note:**
  >
  > If a sharing data source does not specify the PDB name property then `getConnection()` invocations will return a JDBC connection associated with the root container, or the default service of the shared pool configuration.

- You can set the PDB Service Name to specify the name of the service set on the JDBC connection when the connection is repurposed for a specific PDB.

  ```
  weblogic.jdbc.pdbServiceName=<service>
  ```

  > **Note:**
  >
  > The PDB service name attribute is optional. When not specified by the sharing data source configuration a connection will be associated with the default service of the shared pool data source.

- You can set `Proxy User` property to specify the name of the proxy user and password to set on the JDBC connection when the connection is repurposed for the sharing data source.

  ```
  weblogic.jdbc.pdbProxy.<proxy-user>=<proxy-password>
  ```

  > **Note:**
  >
  > Proxy user is only set when a connection is switched to a PDB/service. When both Proxy User and Role Name attributes are specified, the Proxy User takes precedence and the role is not set on the database session.

- You can also set the `Role` property to specify a password-protected role to be set on the JDBC connection when it is repurposed for the sharing data source. There can be any number of password-protected roles configured for a sharing data source.

  ```
  weblogic.jdbc.pdbRole.<role-name>=<role-password>
  ```

  > **Note:**
  >
  > When the `Proxy User` attribute is also set, it takes precedence and the role is not set.

Note:

You can set proxy authentication or password protected roles to secure the sharing data source. For any given shared pool, you can use only one of these options.

Configuring Database for Shared Pooling

To configure your Oracle database to support shared pooling, you need to specify a common database user in the shared pooling data source configuration attributes. This common user must exist in all PDBs that are connected to the sharing data sources.
This common user must have the following privileges:

- grant execute on `dbms_service_prvt to c##user;`
- grant set container to `c##user;`

The shared pooling data source configuration should specify a URL that includes a common service for the CDB.

The password-protected roles need to be defined for the configured common user in each PDB connected to by a sharing data source

Example 8-1 WLST Script for Configuring Shared Pooling

```
import os
def createJDBCSystemResource(owner, resourceName, driver, url, user,
password):
  systemResource=owner.createJDBCSystemResource(resourceName)
  systemResource.setName(resourceName)
  jdbcResource=systemResource.getJDBCResource()
  jdbcResource.setName(resourceName)
  driverParams=jdbcResource.getJDBCDriverParams()
  if driver:
    driverParams.setDriverName(driver)
  if url:
    driverParams.setUrl(dburl)
  properties = driverParams.getProperties()
  if user:
    properties.createProperty('user', user)
  if password:
    driverParams.setPassword(dbpassword)
  return systemResource
def createSharedPoolDS(owner, resourceName, driver, url, user, password):
   systemResource = createJDBCSystemResource(owner, resourceName, driver,
url, user, password)

systemResource.getJDBCResource().getJDBCDriverParams().getProperties().createP
roperty('weblogic.jdbc.sharedPool', 'true')
   return systemResource
def createSharingDS(owner, resourceName, sharedPoolJNDIName, pdbName,
pdbServiceName, roleName, rolePassword):
  systemResource = createJDBCSystemResource(owner, resourceName, driver=None,
url=None, user=None, password=None)

properties=systemResource.getJDBCResource().getJDBCDriverParams().getPropertie
```

```
s()
  properties.createProperty('weblogic.jdbc.sharedPoolJNDIName',
sharedPoolJNDIName)
  if pdbName:
    properties.createProperty("weblogic.jdbc.pdbName", pdbName)
  if pdbServiceName:
    properties.createProperty("weblogic.jdbc.pdbServiceName", pdbServiceName)
  if roleName:
    roleprop=properties.createProperty("weblogic.jdbc.pdbRole."+roleName)
  if rolePassword:
     roleprop.setEncryptedValue(rolePassword)
  return systemResource
servername='myserver'
sharedpoolname='sharedpool'
sharingds1name='sharingds1'
sharingds2name='sharingds2'
driver='oracle.jdbc.OracleDriver'
dburl='jdbc:oracle:thin:@host:1521/orcl'
dbuser='c##1'
dbpassword='xyzzy'
pdb1='pdb1'
pdb1service='coke'
pdb1role='cokerole'
pdb1rolepwd='cokepwd'
pdb2='pdb2'
pdb2service='pepsi'
pdb2role='pepsirole'
pdb2rolepwd='pepsipwd'
connect('weblogic', 'weblogic', 't3://localhost:7001')
edit()
# create shared pool datasource
startEdit()
spds=createSharedPoolDS(cmo, sharedpoolname, driver, dburl, dbuser,
dbpassword)
spds.addTarget(getMBean('/Servers/'+servername))
activate()
startEdit()
# create sharing datasource 1
sharingds1=createSharingDS(owner=cmo, resourceName=sharingds1name,
sharedPoolJNDIName=sharedpoolname, pdbName=pdb1, pdbServiceName=pdb1service,
roleName=pdb1role, rolePassword=pdb1rolepwd)
sharingds1.addTarget(getMBean('/Servers/'+servername))
# create sharing datasource 2
sharingds2=createSharingDS(owner=cmo, resourceName=sharingds2name,
sharedPoolJNDIName=sharedpoolname, pdbName=pdb2, pdbServiceName=pdb2service,
roleName=pdb2role, rolePassword=pdb2rolepwd)
sharingds2.addTarget(getMBean('/Servers/'+servername))
activate()
exit()
```

9 Using Oracle Databases with WebLogic Server

WebLogic Server integrates with specific features of the Oracle database to provide some advanced functionality.

- WebLogic JDBC Features for Oracle Database 23ai
  WebLogic Server 14.1.2.0.0 ships with the Oracle 23ai JDBC drivers. WebLogic 14.1.2.0.0 no longer supports Oracle Database 12c. The purpose of this documentation is to provide historical reference to advanced functionality supported in older versions of the WebLogic Server.
- WebLogic JDBC Features for Oracle Database 19.3
  WebLogic Server 14.1.1.0.0 ships with the Oracle 19.3 JDBC drivers.
- WebLogic JDBC Features for Oracle Database 12.2
  WebLogic JDBC provides several features that specifically require the use of Oracle Database 12.2.x. Learn about supported features with various combinations of WebLogic Server releases and Oracle Database 12.2.x release.
- WebLogic JDBC Features for Oracle Database 12.1
  Learn about Oracle database features supported with various combinations of WebLogic Server, 11g and 12c JDBC drivers, and 11g and 12.1 versions of Oracle Database.

WebLogic JDBC Features for Oracle Database 23ai

WebLogic Server 14.1.2.0.0 ships with the Oracle 23ai JDBC drivers. WebLogic 14.1.2.0.0 no longer supports Oracle Database 12c. The purpose of this documentation is to provide historical reference to advanced functionality supported in older versions of the WebLogic Server.

New features supported by the Oracle 23ai JDBC drivers are:

- Oracle JDBC Driver Extensions
- Support for Oracle DB Sharding with the Native UCP Data Source
- Support for JSON DB support with the JSON driver
- Enhancements to Diagnostic and Logging
- Support Native Boolean datatype

WebLogic JDBC Features for Oracle Database 19.3

WebLogic Server 14.1.1.0.0 ships with the Oracle 19.3 JDBC drivers.

New features supported by the 18c and 19c JDBC drivers are:

- Server Draining ahead of relocating or stopping services or PDB.
- Oracle Database sees Request Boundaries
- Transparent Application Continuity (TAC)

- Support for defining an HTTP proxy host and port in the URL
- Better integration with Autonomous Transaction Processing (ATP) databases

For more information about 18c and 19c JDBC drivers, see *What's New* document in Oracle Database Documentation.

WebLogic Server supports the JDBC 4.3 specifications, when the environment is using JDK 11 and the JDBC driver is JDBC 4.3 compliant. To use new JDBC 4.3 methods, see Support for JDBC 4.3 Interfaces.

- Support for JDBC 4.3 Interfaces

Support for JDBC 4.3 Interfaces

The JDBC 4.3 specification defines new APIs for sharding, unit-of-work demarcation and enquoting literals and identifiers. An overview of the JDBC 4.3 API changes is provided at *https://docs.oracle.com/javase/9/docs/api/java/sql/package-summary.html* .

WebLogic Server supports the new JDBC 4.3 APIs when the environment is using JDK 11 and the JDBC driver is JDBC 4.3 compliant. To use JDBC 4.3 methods with the Oracle thin driver, the 19c or later `ojdbc10.jar` must be on the server CLASSPATH.

The Oracle Application Continuity integration in Generic and Active GridLink data source types internally invoke the unit-of-work demarcation APIs when a connection is reserved from and released back to the connection pool. A new data source configuration attribute can be used to control whether or not automatic unit-of-work demarcation is performed on JDBC connections for Oracle replay drivers and non-Oracle 4.3 drivers that support the APIs. See InvokeBeginEndRequest.

> **Note:**
>
> - The Oracle sharding APIs are only supported with the UCP data source type. See Using Universal Connection Pool Data Sources
> - In WebLogic Server 14.1.2 automatic unit-of-work demarcation will be disabled by default for Generic and Active GridLink data sources that are configured with an Oracle XA driver class.

WebLogic JDBC Features for Oracle Database 12.2

WebLogic JDBC provides several features that specifically require the use of Oracle Database 12.2.x. Learn about supported features with various combinations of WebLogic Server releases and Oracle Database 12.2.x release.

Table 9-1 Oracle Database 12.2 Feature Support

Feature	Description	WebLogic Server Introduced	Database Releases
JDBC 4.2	See JDBC 4.2 Interfaces	12.1.3	12.1.0.1
Service Switching	See Service Switching	12.2.1	12.2

Table 9-1 (Cont.) Oracle Database 12.2 Feature Support

Feature	Description	WebLogic Server Introduced	Database Releases
JDBC Replay Driver	See Database 12.2 JDBC Replay Driver	12.2.1	12.2
UCP MT Shared Pool support	See Universal Connection Pool Multi Tenant Shared Pool support	12.2.1.1.0	12.2
Gradual Draining	During planned maintenance, it is desirable to gradually drain connections instead of closing them all immediately. This prevents uneven performance by the application. See Gradual Draining	12.2.1.2.0	12.1 with 12.2 enhancements
AGL Support for URL with @alias or @ldap	See AGL Support for URL with @alias or @ldap	12.2.1.2.0	12.2
Data Source Shared Pooling	Shared pooling feature provides the ability for multiple data source definitions to share an underlying connection pool. See Using Shared Pooling Data Sources	12.2.1.3.0	12.2
Transaction Guard Integration	Transaction Guard provides a generic infrastructure for applications to use for at-most-once execution during planned and unplanned outages and duplicate submissions. See Using Transaction Guard in *Developing JTA Applications for Oracle WebLogic Server*.	12.2.1.3.0	12.1.0.2

WebLogic JDBC features for Oracle Database 12.2 are:

- JDBC 4.2 Interfaces
- Database 12.2 JDBC Replay Driver
- AGL Support for URL with @alias or @ldap

JDBC 4.2 Interfaces

JDK 8 has new API's for JDBC 4.2 that are supported for any database driver that is JDBC 4.2 compliant. The first Oracle driver to support JDK 8 and JDBC 4.2 is 12.2.0.1

The following are the features introduced in JDBC 4.2 java.sql and javax.sql

- Added `JDBCType` enum and `SQLType` interface
- Support for `REF CURSORS` in `CallableStatement`
- `DatabaseMetaData` methods to return maximum `Logical LOB` size and if Ref Cursors are supported
- Added support for large update counts

The JDBC 4.2 API changes are documented at https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/jdbc_42.html

Database 12.2 JDBC Replay Driver

The JDBC Replay Driver is new in Oracle Database 12.2. The name is `oracle.jdbc.replay.OracleXADataSourceImpl`. If the WebLogic Server is run on a driver earlier than 12.2, an error will be thrown indicating that the class cannot be loaded.

If the data source is running with the JDBC Replay Driver from database 12.2, the test table name is validated as follows:

- If `SQL ISVALID` or `SQL PINGDATABASE`, no change.
- If a table name (which is converted to `select count(*)` from tablename) or `SQL SELECT` is specified, it is converted to `SQL ISVALID`.
- Any other value (DML or DDL) will cause an exception to be thrown and the data source will not deploy.

Note:

12.2 JDBC Replay Driver does not support replay with global transactions, it supports local transactions on an XA connection.

AGL Support for URL with @alias or @ldap

This feature allows for using an alias or an LDAP connection in the AGL URL. The alias format is `jdbc:oracle:thin:@alias` where "alias" is an alias defined in a `tnsnames.ora` file.

See LDAP Syntax.

To use an alias, you need to you need to perform the following steps:

1. Specify the system property -`Doracle.net.tns_admin=tns_directory`, where `tns_directory`, is the directory location of the `tnsnames.ora` file.
2. Create or modify a `tnsnames.ora` file in the directory location specified by `tns_directory`.

   The entry has the form: `alias=(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=host)(PORT=port))(CONNECT_DATA=(SERVICE_NAME=service)))`

   where host is URL of a database listener, port is the port a database listener, and service is the service name of the database you would like to connect to.
3. Use the alias in the data source definition URL by replacing the connection string with the alias.

WebLogic JDBC Features for Oracle Database 12.1

Learn about Oracle database features supported with various combinations of WebLogic Server, 11g and 12c JDBC drivers, and 11g and 12.1 versions of Oracle Database.

Table 9-2 Oracle Database 12.1 Feature Support

Feature	WebLogic Server 10.3.6/12.1.x with 11g drivers and Oracle Database 11gR2	WebLogic Server 10.3.6/12.1.x with 11g drivers and Oracle Database 12c	WebLogic Server 10.3.6/12.1.1 with 12c and later drivers and Oracle Database 11gR2	WebLogic Server 12.1.2 and later with 12c and later drivers and Oracle Database 11gR2	WebLogic Server 10.3.6/12.1.1 with 12c and later drivers and Oracle Database 12c	WebLogic Server 12.1.2 and later with 12c and later drivers with Oracle Database 12c
JDBC Replay Driver (read/ write)	No	No	No	No	Yes (Read/ Write with Active GridLink only, no XA transactions)	Yes (Read/ Write with Active GridLink and Generic data source, no XA transactions)
Pluggable Database (PDB)	No	Yes (Except `Set Container`)	No	No	Yes	Yes
Dynamic switching between PDBs	No	No	No	No	No	Yes (no XA)
Database Resident Connection Pooling(DRCP)	No	No	No	Yes	No	Yes
Oracle Notification Service (ONS) auto configuration	No	No	No	No	No	Yes (Active GridLink only)
Global Data Services (GDS)	No	Yes (Active GridLink only)	No	No	Yes (Active GridLink only)	Yes (Active GridLink only)
JDBC 4.1 (using `ojdbc7.jar` files and JDK 7)	No	No	Yes	Yes	Yes	Yes

WebLogic JDBC features for Oracle Database 12.1 are:

- JDBC 4.1 Support for JDK 7
- JDBC Replay Driver Support
- Database Resident Connection Pooling Support
- Container Database with Pluggable Databases
- Global Data Services Support
- Automatic ONS Listeners

JDBC 4.1 Support for JDK 7

WebLogic Server supports the JDBC 4.1 specification when the environment is using JDK 7 and the JDBC driver is JDBC 4.1 compliant. To use new JDBC 4.1 methods, you must use the `ojdbc7.jar`. See JDBC™ 4.1 Specification.

WebLogic Server currently does not support the `java.sql.driver` interfaces required to use the Java SE 8 `getParentLogger` method. See `http://docs.oracle.com/javase/8/docs/api/index.html?java/sql/Driver.html`.

JDK 7 also brings support for minor changes in Rowset 1.1 defined at `http://jcp.org/aboutJava/communityprocess/maintenance/jsr114/114MR2approved.pdf`. The WebLogic Server implementation of the new `RowSetFactory` is called `weblogic.jdbc.rowset.JdbcRowSetFactory`.

JDBC Replay Driver Support

JDBC Replay Driver (also referred to as Replay) is a general purpose, application-independent infrastructure for Active GridLink and Generic data sources that enables the recovery of work from an application perspective and masks many system, communication, and hardware failures. See JDBC Replay Driver.

Database Resident Connection Pooling Support

Database Resident Connection Pooling (DRCP) is an Oracle database server feature that provides the ability to share connections among multiple connection pools that can span across mid-tier systems. See Database Resident Connection Pooling.

Container Database with Pluggable Databases

Container Database (CDB) is an Oracle Database feature that minimizes the overhead of having many of databases by consolidating them into a single database with multiple Pluggable Databases (PDB) in a single CDB. See Container Database with Pluggable Databases.

Global Data Services Support

Global Data Services (GDS) is an Oracle Database feature that provides automated load balancing, fault tolerance and resource utilization in a distributed database environment. See Global Database Services.

Automatic ONS Listeners

If you are using Oracle Database 12c with WebLogic Server version 12.1.2 or later, you are no longer required to provide the ONS Listener list as part of an Active GridLink data source configuration. The ONS list is automatically provided from the database to the driver. See Enabling FAN Events.

10 Labeling Connections

Label connections increase the performance of database connections. By associating particular labels with particular connection states, an application can retrieve an already initialized connection from the pool and avoid the time and cost of re-initialization.

- What is Connection Labeling
  Applications often initialize connections retrieved from a connection pool before using the connection. The initialization varies and could include simple state re-initialization that requires method calls within the application code or database operations that require round trips over the network. The cost of such initialization may be significant. Labeling connections allows an application to attach arbitrary name/value pairs to a connection.
- Implementing Labeling Callbacks
  A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Applications that use the connection labeling feature must provide a callback implementation.
- Creating a Labeling Callback
  A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Learn how to create a labeling callback by implementing the `oracle.ucp.ConnectionLabelingCallback` interface.
- Registering a Labeling Callback
  A WebLogic Server data source provides the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. Only one callback may be registered on a connection pool.
- Reserving Labeled Connections
  WebLogic JDBC data sources provide two `getConnection` methods that are used for reserving a labeled connection from the pool.
- Checking Unmatched labels
  Connections may have multiple labels, which each uniquely identify the connection based on specified criteria. Use the `getUnmatchedConnectionLabels` method to verify which connections do not match the requested label.
- Removing a Connection Label
  You can remove a connection label by using the `removeConnectionLabel` method.
- Using Initialization and Reinitialization Costs to Select Connections
- Using Connection Labeling with Packaged Applications
  WebLogic Server allows callbacks, such as connection labeling and connection initialization, in EAR or WAR files used by a packaged application.

What is Connection Labeling

Applications often initialize connections retrieved from a connection pool before using the connection. The initialization varies and could include simple state re-initialization that requires method calls within the application code or database operations that require round trips over

the network. The cost of such initialization may be significant. Labeling connections allows an application to attach arbitrary name/value pairs to a connection.

The application can request a connection with the desired label from the connection pool. The connection labeling feature does not impose any meaning on user-defined keys or values; the meaning of user-defined keys and values is defined solely by the application.

Some of the examples for connection labeling include role, NLS language settings, transaction isolation levels, stored procedure calls, or any other state initialization that is expensive and necessary on the connection before work can be executed by the resource.

Connection labeling is application-driven and requires the following:

- The `oracle.ucp.jdbc.LabelableConnection` interface is used to apply and remove connection labels, as well as retrieve labels that have been set on a connection.
- The `oracle.ucp.ConnectionLabelingCallback` interface is used to create a labeling callback that determines whether or not a connection with a requested label already exists. If no connections exist, the interface allows current connections to be configured as required.
- A `Connection Labeling Callback`.

Implementing Labeling Callbacks

A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Applications that use the connection labeling feature must provide a callback implementation.

A labeling callback is used when a labeled connection is requested but there are no connections in the pool that match the requested labels. The callback determines which connection requires the least amount of work in order to be re-configured to match the requested label and then allows the connection's labels to be updated before returning the connection to the application.

Note:

Connection Labeling is not supported from client applications that use RMI. See Using the WebLogic RMI Driver (Deprecated) in *Developing JDBC Applications for Oracle WebLogic Server*.

Creating a Labeling Callback

A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Learn how to create a labeling callback by implementing the `oracle.ucp.ConnectionLabelingCallback` interface.

To create a labeling callback, an application implements the `oracle.ucp.ConnectionLabelingCallback` interface. One callback is created per connection pool. The interface provides two methods as shown below:

```
public int cost(Properties requestedLabels, Properties currentLabels);

public boolean configure(Properties requestedLabels, Connection conn);
```

The connection pool iterates over each connection available in the pool. For each connection, it calls the `cost` method. The result of the `cost` method is an integer which represents an estimate of the cost required to reconfigure the connection to the required state. The larger the value, the costlier it is to reconfigure the connection. The connection pool always returns connections with the lowest cost value. The algorithm is as follows:

- If the cost method returns 0 for a connection, the connection is a match (note that this does not guarantee that `requestedLabels` equals `currentLabels`). The connection pool does not call configure on the connection found and simply returns the connection.
- If the cost method returns a value that is not 0 (a negative or positive integer), then the connection pool iterates until it finds a connection with a cost value of 0 or runs out of available connections.
- If the pool has iterated through all available connections and the lowest cost of a connection is `Integer.MAX_VALUE` (2147483647 by default), then no connection in the pool is able to satisfy the connection request. The pool creates a new connection, calls the `configure` method on it, and then returns this new connection. If the pool has reached the maximum pool size (it cannot create a new connection), then the pool either throws an SQL exception or waits if the connection wait timeout attribute is specified.
- If the pool has iterated through all available connections and the lowest cost of a connection is less than `Integer.MAX_VALUE`, then the configure method is called on the connection and the connection is returned. If multiple connections are less than `Integer.MAX_VALUE`, the connection with the lowest cost is returned.

There is also an extended callback interface `oracle.ucp.jdbc.ConnectionLabelingCallback` that has an additional `getRequestedLabels()` method. `getRequestedLabels` is invoked at `getConnection()` time when no requested labels are provided and there is an instance registered. This occurs when the standard `java.sql.Datasource getConnection()` methods are used that do not provide the label information on the `getConnection()` call.

- Example Labeling Callback

Example Labeling Callback

The following code example demonstrates a simple labeling callback implementation that implements both the `cost` and `configure` methods. The callback is used to find a labeled connection that is initialized with a specific transaction isolation level.

Example 10-1 Labeling Callback

```
import oracle.ucp.jdbc.ConnectionLabelingCallback;
import oracle.ucp.jdbc.LabelableConnection;
import java.util.Properties;
import java.util.Map;
import java.util.Set;
import weblogic.jdbc.extensions.WLDataSource;
class MyConnectionLabelingCallback implements ConnectionLabelingCallback {

  public MyConnectionLabelingCallback()  {
  }
  public int cost(Properties reqLabels, Properties currentLabels)  {
    // Case 1: exact match
    if (reqLabels.equals(currentLabels)) {
      System.out.println("## Exact match found!! ##");
      return 0;
    }

   // Case 2: some labels match with no unmatched labels
```

```
    String iso1 = (String) reqLabels.get("TRANSACTION_ISOLATION");
    String iso2 = (String) currentLabels.get("TRANSACTION_ISOLATION");
    boolean match =
      (iso1 != null && iso2 != null && iso1.equalsIgnoreCase(iso2));
    Set rKeys = reqLabels.keySet();
    Set cKeys = currentLabels.keySet();
    if (match && rKeys.containsAll(cKeys)) {
      System.out.println("## Partial match found!! ##");
      return 10;

    }
    // No label matches to application's preference.
    // Do not choose this connection.
    System.out.println("## No match found!! ##");
    return Integer.MAX_VALUE;
  }

 public boolean configure(Properties reqLabels, Object conn)  {
    try {
      String isoStr = (String) reqLabels.get("TRANSACTION_ISOLATION");
      ((Connection)conn).setTransactionIsolation(Integer.valueOf(isoStr));
      LabelableConnection lconn = (LabelableConnection) conn;

     // Find the unmatched labels on this connection
      Properties unmatchedLabels =
           lconn.getUnmatchedConnectionLabels(reqLabels);
      // Apply each label <key,value> in unmatchedLabels to conn

     for (Map.Entry<Object, Object> label : unmatchedLabels.entrySet())  {
        String key = (String) label.getKey();
        String value = (String) label.getValue();

        lconn.applyConnectionLabel(key, value);

      }
    } catch (Exception exc) {
      return false;
    }
    return true;
  }

public java.util.Properties getRequestedLabels() {
    Properties props = new Properties();

 // Set based on some application state that might be on a thread-local, http session
info, etc.
    String value = "value";


  props.put("TRANSACTION_ISOLATION", value);

  return props;
  }
}
```

Registering a Labeling Callback

A WebLogic Server data source provides the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. Only one callback may be registered on a connection pool.

See, the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. The following code example demonstrates registering a labeling callback that is implemented in the `MyConnectionLabelingCallback` class:

```
. . .
import weblogic.jdbc.extensions.WLDataSource;
. . .
MyConnectionLabelingCallback callback = new MyConnectionLabelingCallback();
((WLDataSource)ds).registerConnectionLabelingCallback( callback );
. . .
```

You can also register the callback using the WebLogic Remote Console.

- Removing a Labeling Callback
- Applying Connection Labels

Removing a Labeling Callback

You can remove a labeling callback by using one of the following methods:

- If you have programmatically set a callback, use the `removeConnectionLabelingCallback()` method as shown in the following example:

  ```
  . . .
  import weblogic.jdbc.extensions.WLDataSource;
  . . .
  ((WLDataSource)ds).removeConnectionLabelingCallback( callback );
  . . .
  ```

- If you have administratively configured the callback, remove the callback from the data source configuration.

Note:

An application must use one of the methods to register and remove callbacks but not both. For example, if you register the callback on a connection using `registerConnectionLabelingCallback(callback)`, you must use `removeConnectionLabelingCallback()` to remove it.

Applying Connection Labels

Labels are applied on a reserved connection using the `applyConnectionLabel` method from the `LabelableConnection` interface. Any number of connection labels may be cumulatively applied on a reserved connection. Each time a label is applied to a connection, the supplied key/value pair is added to the existing collection of labels. Only the last applied value is retained for any given key.

Note:

A labeling callback must be registered on the connection pool before a label can be applied on a reserved connection; otherwise, labeling is ignored. See Creating a Labeling Callback.

The following example demonstrates initializing a connection with a transaction isolation level and then applying a label to the connection:

```
. . .
String pname = "property1";
String pvalue = "value";
Connection conn = ds.getConnection();
// initialize the connection as required.
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
((LabelableConnection) conn).applyConnectionLabel(pname, pvalue);
. . .
```

Reserving Labeled Connections

WebLogic JDBC data sources provide two `getConnection` methods that are used for reserving a labeled connection from the pool.

The syntax of the two methods is:

- Public Connection `getConnection(java.util.Properties labels)`
- Public Connection `getConnection(String user, String password, java.util.Properties labels)`

The methods require that the label be passed to the `getConnection` method as a `Properties` object. The following example demonstrates getting a connection with the label *property1* value.

```
. . .
import weblogic.jdbc.extensions.WLDataSource;
. . .
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);
. . .
Connection conn = ((WLDataSource)ds).getConnection(label);
. . .
```

It is also possible to use the standard `java.sql.Datasource getConnection()` methods. In this case, the label information is not provided on the `getConnection()` call. The interface `oracle.ucp.jdbc.ConnectionLabelingCallback` is used:

```
java.util.Properties getRequestedLabels();
```

`getRequestedLabels` is invoked at `getConnection()` time when no requested labels are provided and there is an instance registered.

Checking Unmatched labels

Connections may have multiple labels, which each uniquely identify the connection based on specified criteria. Use the `getUnmatchedConnectionLabels` method to verify which connections do not match the requested label.

This method is used after a connection with multiple labels is reserved from the connection pool and is typically used by a labeling callback. See`getUnmatchedConnectionLabels` method.

The following code example demonstrates checking for unmatched labels:

```
. . .
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);
. . .
Connecion conn = ((WLDataSource)ds).getConnection(label);
Properties unmatched =
   ((LabelableConnection)connection).getUnmatchedConnectionLabels (label);
. . .
```

Removing a Connection Label

You can remove a connection label by using the `removeConnectionLabel` method.

This method is used after a labeled connection is reserved from the connection pool. See `removeConnectionLabel`.

The following code example demonstrates removing a connection label:

```
. . .
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);
Connection conn = ((WLDataSource)ds).getConnection(label);
. . .
((LabelableConnection) conn).removeConnectionLabel(pname);
. . .
```

Using Initialization and Reinitialization Costs to Select Connections

Some applications require that a connection pool be able to identify high-cost connections and avoid using those connections when the number of connections is below a certain threshold. Using that information allows a connection pool to use new physical connections to serve connection requests from different tenants without incurring reinitialization overhead on other tenant connections already in the pool.
WebLogic Server provides the following connection properties to identify high cost connections:

- `ConnectionLabelingHighCost`—When greater than 0, connections with a cost value equal to or greater than the property value are considered *high-cost* connections. The default value is `Integer.MAX_VALUE`.

For example, if the property value is set to 5, any connection whose calculated cost value from the labeling callback is equal to or greater than 5 is considered a *high-cost* connection.

- `HighCostConnectionReuseThreshold`—When greater than 0, specifies a threshold of the number of total connections in the pool beyond which Connection Labeling is allowed to reuse *high-cost* connections in the pool to serve a request. Below this threshold, Connection Labeling either uses an available low-cost connection or creates a brand-new physical connection to serve a request. The default value is 0.

  For example, if set to 20, Connection Labeling reuses *high-cost* connections when there are no low-cost connections available and the total connections reach 20.

- For Generic data sources see Configuring Generic Connection Pool Features.
- For Active GridLink data sources, see Configuring AGL Connection Pool Features.
- Considerations When Using Initialization and Reinitialization Costs

Considerations When Using Initialization and Reinitialization Costs

This section provides additional considerations when selecting connections based on connection costs:

- Valid callback registration activates Connection Labeling. Once registered, the connection pool checks for new threshold values at regular intervals and determines:
  - if a connnection has a cost that is equal to or greater than `ConnectionLabelingHighCost`.
  - If the number of total connections accounts for the number of active connection creation requests, including restrictions for `MinCapacity` and `MaxCapacity`.
- Any labeled connection with cost value of `Integer.MAX_VALUE` is not reused, even if a new threshold is reached.
- There is no requirement not to reuse connections without labels (stateless) in the pool to serve connection requests with labels (labeled requests). Once the `HighCostConnectionReuseThreshold` is reached and Connection Labeling is activated, the pool continues to favor connections without labels (stateless) over creating new physical connections.

Using Connection Labeling with Packaged Applications

WebLogic Server allows callbacks, such as connection labeling and connection initialization, in EAR or WAR files used by a packaged application.

To define an application-packaged callback class in a data source configuration:

- Define the data source as part of the application.

  For example, if the callback implementation classes are packaged in a WAR or defined as part of a shared library that is referenced by the application, the EAR file contains application packaged data source configurations that reference the callback class names in their module descriptors.

- Specify the application WAR file (that contains the callback implementations) as part of the application `classloader` hierarchy in the `weblogic-application.xml` file.

  For example:

```
. . .
<classloader-structure>
  <module-ref>
    <module-uri>appcallbacks.war</module-uri>
  </module-ref>
</classloader-structure>
```

Considerations When using Labelled Connections in Packaged Applications

WebLogic Server does not support specifying a connection labeling callback or connection initialization callback in the module descriptor for a globally scoped data source system resource when the callback class is packaged in an application. A global data source requires that callback implementation classes be on the WebLogic classpath. However, you can workaround this restriction for an application callback that is packaged in a WAR or EAR by having the application register the callback at runtime using the WLDataSource interface in the *Java API Reference for Oracle WebLogic Server*.

11 Understanding Data Source Security

Secure WebLogic JDBC data sources by configuring the data source security options in your application environment. Security considerations include the number of WebLogic Server and database users, the granularity of data access, the depth of the security identity (property on the connection or a real user), performance, coordination of various components in the software stack, and driver capabilities.

- About WebLogic Data Source Security Options
  By default, you define a single database user and password for a data source. You can store it in the data source descriptor or make use of the wallet.
- WebLogic Data Source Security Options
  Learn about the security options available for WebLogic JDBC data source.
- Connections within Transactions
  When you get a connection within a transaction, it is associated with the transaction context on a particular WebLogic Server instance. This type of connection has some special behaviors.
- WebLogic Data Source Resource Permissions
- Data Source Security Example
  Learn about the interactions and differences between Identity, Proxy, and Database Credentials with help of data source security example.
- Using Encrypted Connection Properties
- Using SSL and Encryption with Data Sources and Oracle Drivers
  Use SSL to provide both data encryption and strong authentication for network connections to the database server. This topic provides additional information on using these features with WebLogic Server.

About WebLogic Data Source Security Options

By default, you define a single database user and password for a data source. You can store it in the data source descriptor or make use of the wallet.

For information on using wallets, see Creating and Managing Oracle Wallet. This is a very simple and efficient approach to security. All of the connections in the connection pool are owned by this user and there is no special processing when a connection is given out. That is, it's a homogenous connection pool and any request can get any connection from a security perspective (there are other aspects, such as affinity). Regardless of the end user of the application, all connections in the pool use the same security credentials to access the DBMS. No additional information is needed when you get a connection because it's all available from the data source descriptor or wallet. For example:

```
java.sql.Connection conn =  mydatasource.getConnection();
```

Note:

You can enter the password as a name-value pair in the `Properties` field (this 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.

It is recommended 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 WebLogic Remote Console. Also, `JDBCDriverParamsBean.Password` attribute is now dynamic and does not require a restart of the data source.

The JDBC API can also be used to programmatically specify the database username and password as in the following.

```
java.sql.Connection conn = mydatasource.getConnection("user", "password");
```

Although the JDBC specification implies that the `getConnection("user", "password")` method should take a database user and associated password, software vendors have developed implementations according to their own interpretation of the specification. Oracle WebLogic Server, by default, treats this as an application server user and password:

- The pair is authenticated to see if it is a valid user and that user is used for WebLogic security permission checks.
- The user is then mapped to a database user and password using the data source credential mapper.

WebLogic Server's implementation generically follows the specification but the database credentials are one-step removed from the application code.

While the default approach is simple, it does mean that only one user is doing all of the work. You can't determine who actually did the update nor can you restrict SQL operations by who is running the operation, at least at the database level. Any type of per-user logic needs to be in the application code instead of relying on the database. There are various WebLogic data source features that can be configured to provide per-user information about the operations.

WebLogic Data Source Security Options

Learn about the security options available for WebLogic JDBC data source.

Table 11-1 WebLogic Data Source Configuration Options for Security Credentials

Feature	Description	Can be used with . . .	Can't be used with . . .
User authentication (default)	Default `getConnection(user, password)` behavior – WebLogic validates the input and uses the user/ password in the descriptor.	Proxy session, Set client identifier	Identity pooling, Use database credentials

Table 11-1 (Cont.) WebLogic Data Source Configuration Options for Security Credentials

Feature	Description	Can be used with . . .	Can't be used with . . .
Use database credentials	Instead of using the credential mapping, use the supplied user and password directly.	Set client identifier, Proxy session, Identity pooling	User authentication
Set Client Identifier	Set a client identifier property associated with the connection (Oracle and DB2 only).	Everything	N/A
Proxy Session	Set a light-weight proxy user associated with the connection (Oracle-only).	User authentication, Set client identifier, Use database credentials	Identity pooling
Identity pooling	Heterogeneous pool of connections owned by specified users.	Set client identifier, Use database credentials	Proxy session, User authentication, Labeling, Active GridLink

Note:

All of these features are available with both XA and non-XA drivers.

- Credential Mapping vs. Database Credentials
- Set Client Identifier on Connection
- Oracle Proxy Session
- Identity-based Connection Pooling

Credential Mapping vs. Database Credentials

Each WebLogic data source has a credential map that is a mechanism used to map a key, in this case a WebLogic user, to security credentials (user and password). By default, when a user and password are specified when getting a connection, they are treated as credentials for a WebLogic user, validated, and are converted to a database user and password using a credential map associated with the data source. If a matching entry is not found in the credential map for the data source, then the user and password associated with the data source definition are used. Because of this defaulting mechanism, you should be careful what permissions are granted to the default user. Alternatively, you can define an invalid default user to ensure that no one can accidentally get through (in this case, you would need to set the initial capacity for the pool to zero so that the pool is populated only by valid users).

To create an entry in the credential map:

1. Create a WebLogic user. In the WebLogic Remote Console, go to Security realms, select your realm (for example, myrealm), and select New.
2. Create the mapping. See *Add a Credential Mapping* in *Oracle WebLogic Remote Console Online Help*.

The advantages of using the credential mapping are that:

- You don't hard-code the database user/password into a program or need to prompt for it in addition to the WebLogic user/password.
- It provides a layer of abstraction between WebLogic security and database settings such that many WebLogic identities can be mapped to a smaller set of DB identities, thereby only requiring middle-tier configuration updates when WebLogic users are added/removed.

You can cut down the number of users that have access to a data source to reduce the user maintenance overhead. For example, suppose that a servlet has the one pre-defined WebLogic user/password for data source access that is hardwired in its code using a `getConnection(`*user*`,` *password*`)` call. Every WebLogic user can reap the specific DBMS access coded into the servlet, but none has to have general access to the data source. For instance, there may be a *Sales* DBMS which needs to be protected from unauthorized eyes, but it contains some day-to-day data that everyone needs. The *Sales* data source is configured with restricted access and a servlet is built that hardwires the specific data source access credentials in its connection request. It uses that connection to deliver only the generally needed day-to-day info to any caller. The servlet cannot reveal any other data and no WebLogic user can get any other access to the data source. This is the approach that many large applications use and is the logic behind the default mapping behavior in WebLogic Server.

The disadvantages of using the credential map are that:

- It is difficult to manage (create, update, delete) with a large number of users; it is possible to use WLST scripts or a custom JMX client utility to manage credential map entries.
- You can't share a credential map between data sources so they must be duplicated.

Some applications prefer not to use the credential map. Instead, the credentials passed to `getConnection(`*user*`,` *password*`)` should be treated as database credentials and used to authenticate with the database for the connection, avoiding going through the credential map. This is enabled by setting the `use-database-credentials` to `true`.

When `use-database-credentials` is enabled, it turns of credential mapping for the following attributes:

- `identity-based-connection-pooling-enabled`
- `oracle-proxy-session`
- `set client identifier`

> **Note:**
>
> in the data source schema, the set client identifier feature is poorly named `credential-mapping-enabled`. The documentation and the console refer to it as set client identifier.).

To review the behavior of credential mapping and using database credentials:

- If using the credential map, there needs to be a mapping for each WebLogic user to database user for those users that have access to the database; otherwise the default user for the data source is used. If you always specify a user/password when getting a connection, you only need credential map entries for those specific users.
- If using database credentials without specifying a user/password, the default user and password in the data source descriptor are always used. If you specify a user/password

when getting a connection, that user is used for the credentials. WebLogic users are not involved at all in the data source connection process.

Set Client Identifier on Connection

When this feature is enabled on the data source, a client property is associated with the connection. The underlying SQL user remains unchanged for the life of the connection but the client value can change. This information can be used for accounting, auditing, or debugging. The `client` property is based on either the WebLogic user mapped to a database user based on the credential map or the database user parameter directly from the `getConnection()` method, based on the *use database credentials* setting described earlier.

To enable this feature, select `Set Client ID On Connection` in the WebLogic Remote Console.

The Set Client Identifier feature is only available for use with the Oracle thin driver and the IBM DB2 driver, based on the following interfaces:

- For pre-Oracle 12c, `oracle.jdbc.driver.OracleConnection.setClientIdentifier(client)` is used. For more information about how to use this for auditing and debugging, see Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity in the *Oracle Database Security Guide*. You can get the value using `getClientIdentifier()` from the driver using the `ojdbcN.jar` or `ojdbcN_g.jar` files.

  > **Note:**
  >
  > Setting the client identifier using the Oracle driver is disabled if you are using `ojdbcNdms.jar`, the default JAR file for Oracle Fusion MiddleWare and Oracle Fusion Applications. In this case, the Set Client Identifier feature is not supported.

  To get back the value from the database as part of a SQL query, use a statement like the following:

  ```
  select sys_context('USERENV','CLIENT_IDENTIFIER') from DUAL
  ```

- Starting in Oracle 12c, `java.sql.Connection.setClientInfo("OCSID.CLIENTID", client)` is used. This is a JDBC standard API, although the property values are proprietary. A problem with `setClientIdentifier` usage is that there are pieces of the Oracle technology stack that set and depend on this value. If application code also sets this value, it can cause problems. This has been addressed with `setClientInfo` by making use of this method a privileged operation. A well-managed container can restrict the Java security policy grants to specific namespaces and code bases, and protect the container from out-of-control user code. When running with the Java security manager, permission must be granted in the Java security policy file for:

  ```
  permission "oracle.jdbc.OracleSQLPermission" "clientInfo.OCSID.CLIENTID";
  ```

  Using the name `OCSID.CLIENTID` allows for upward compatible use of `select sys_context('USERENV','CLIENT_IDENTIFIER') from DUAL` or use the JDBC standard API `java.sql.getClientInfo("OCSID.CLIENTID")` to retrieve the value.

- Setting this value in the Oracle `USERENV` context can be used to drive the Oracle Virtual Private Database (VPD) feature to create security policies to control database access at

the row and column level. Essentially, Oracle Virtual Private Database adds a dynamic `WHERE` clause to a SQL statement that is issued against the table, view, or synonym to which an Oracle Virtual Private Database security policy was applied. See Using Oracle Virtual Private Database to Control Data Access in the *Oracle Database Security Guide*. Using this data source feature means that no programming is needed on the WebLogic side to set this context. The context is set and cleared by the WebLogic data source code.

- For the IBM DB2 driver, `com.ibm.db2.jcc.DB2Connection.setDB2ClientUser(client)` is used for older releases (prior to version 9.5). This specifies the current client user name for the connection. Note that the current client user name can change during a connection (unlike the user). This value is also available in the `CURRENT CLIENT_USERID` special register. You can select it using a statement like `select CURRENT CLIENT_USERID from SYSIBM.SYSTABLES`.
- When running the IBM DB2 driver with JDBC 4.0 (starting with version 9.5), `java.sql.Connection.setClientInfo("ClientUser", client)` is used. You can retrieve the value using `java.sql.Connection.getClientInfo("ClientUser")` instead of the DB2 proprietary API (even if set using `setDB2ClientUser()`).

Oracle Proxy Session

Oracle proxy authentication allows one JDBC connection to act as a proxy for multiple (serial) light-weight user connections to an Oracle database with the thin driver. You can configure a WebLogic data source to allow a client to connect to a database through an application server as a proxy user. The client authenticates with the application server and the application server authenticates with the Oracle database. This allows the client's user name to be maintained on the connection with the database.

> **Note:**
>
> This feature is only supported when using the Oracle thin driver and a supported Oracle database (the database URL must contain `oracle`).

Use the following steps to configure proxy authentication on a connection to an Oracle database.

1. If you have not yet done so, create the necessary database users.
2. On the Oracle database, provide `CONNECT THROUGH` privileges. For example:

   ```
   SQL> ALTER USER connectionuser GRANT CONNECT THROUGH dbuser;
   ```

   where `connectionuser` is the name of the application user to be authenticated and `dbuser` is an Oracle database user.
3. Create a Generic or Active GridLink data source and set the user to the value of `dbuser`.
4. To use:
   - WebLogic credentials, create an entry in the credential map that maps the value of `wlsuser` to the value of `dbuser`, as described earlier.
   - Database credentials, enable **Use Database Credentials**, as described earlier.
5. Enable Oracle Proxy Authentication.
6. Log on to a WebLogic Server instance using the value of `wlsuser` or `dbuser`.

7. Get a connection using `getConnection(username, password)`. The credentials are based on either the WebLogic user that is mapped to a database user or the database user directly, based on the **Use Database Credentials** setting.

You can see the current user and proxy user by executing:

```
select user, sys_context('USERENV','PROXY_USER') from DUAL
```

Note:

`getConnection` fails if **Use Database Credentials** is not enabled and the value of the user/password is not valid for a WebLogic user. Conversely, it fails if **Use Database Credentials** is enabled and the value of the user/password is not valid for a database user.

A proxy session is opened on the connection based on the user each time a connection request is made on the pool. The proxy session is closed when the connection is returned to the pool. Opening or closing a proxy session has the following impact on JDBC objects:

- Closes any existing statements (including result sets) from the original connection.
- Clears the WebLogic Server statement cache.
- Clears the client identifier, if set.
- The WebLogic Server test statement for a connection is recreated for every proxy session.

These behaviors may impact applications that share a connection across instances and expect some state to be associated with the connection.

Oracle proxy session is also implicitly enabled when **Use Database Credentials** is enabled and `getConnection(user, password)` is called.

The exact definition of `oracle-proxy-session` is as follows:

- If proxy authentication is enabled and identity based pooling is also enabled, it is an error.
- If a user is specified on `getConnection()` and `identity-based-connection-pooling-enabled` is `false`, then `oracle-proxy-session` is treated as `true` implicitly (it can also be explicitly `true`).
- If a user is specified on `getConnection()` and `identity-based-connection-pooling-enabled` is `true`, then `oracle-proxy-session` is treated as `false`.
- By default, if no credential mapper entry exists for the current user, or the user specified in the `getConnection(user, password)` operation, then the `getConnection()` call will throw a `SQLException` indicating an invalid proxy user.
- To allow the `getConnection()` operation to return a connection with datasource configured credentials, when no credential mapper entry exists, it is necessary to grant the database permission `ALTER USER dbuser GRANT CONNECT THROUGH dbuser`, where `dbuser` is the database user specified in the datasource configuration.
- The **LoggingLastResource** and **TwoPhaseCommit** datasource transaction options require the above `CONNECT THROUGH` privileges for the configured datasource user.

Identity-based Connection Pooling

An identity based pool creates a heterogeneous pool of connections. This allows applications to use a JDBC connection with a specific DBMS credential by pooling physical connections with different DBMS credentials. The DBMS credential is based on either the WebLogic user mapped to a database user or the database user directly, based on the `use-database-credentials`. `use-database-credentials=true` is how some implementations interpret the JDBC standard—basically a heterogeneous pool with users specified by `getConnection(`*user*`,`*password*`)`.

The allocation of connections is more complex if `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.

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

1. At connection pool initialization, the physical JDBC connections based on the configured or default “initial capacity" are created with the configured default DBMS credential of the data source.
2. An application tries to get a connection from a data source.
3. If:
   - `use-database-credentials` is not enabled, the user specified in `getConnection` is mapped to a DBMS credential, as described earlier. If the credential map doesn't have a matching user, the default DBMS credential is used from the data source descriptor.
   - `use-database-credentials` is enabled, the user and password specified in `getConnection` are used directly.
4. The connection pool is searched for a connection with a matching DBMS credential.
5. If a match is found, the connection is reserved and returned to the application.
6. 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.

It should be clear that finding a matching connection is more expensive than a homogeneous pool. Destroying a connection and getting a new one is very expensive. If possible, use a normal homogeneous pool or one of the light-weight options (client identity or an Oracle proxy connection) as they are more efficient than identity-based pooling.

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.

To configure this feature, select `Enable Identity Based Connection Pooling`.

You must make the following changes to use Logging Last Resource (LLR) transaction optimization with Identity-based Pooling to get around the problem that multiple users access the associated transaction table:

- 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 that could access this table via a global transaction. 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.

Connections within Transactions

When you get a connection within a transaction, it is associated with the transaction context on a particular WebLogic Server instance. This type of connection has some special behaviors.

For example:

- When getting a connection with a data source configured with non-XA LLR or 1PC (JTS driver) with global transactions, the first connection obtained within the transaction is returned on subsequent connection requests regardless of the values of username/ password specified and independent of the associated proxy user session, if any. The connection must be shared among all users of the connection when using LLR or 1PC.
- For XA data sources, the first connection obtained within the global transaction is returned on subsequent connection requests within the application server, regardless of the values of username/password specified and independent of the associated proxy user session, if any. The connection must be shared among all users of the connection within a global transaction within the application server/JVM.

WebLogic Data Source Resource Permissions

In WebLogic Server, security policies answer the question "who has access" to a WebLogic data source resource. A security policy is created when you define an association between a WebLogic data source resource and a user, group, or role. You can optionally restrict access to JDBC data sources using security policies.
A WebLogic data source resource has no protection until you assign it a security policy. As soon as you add one policy for a permission, then all other users are restricted. For example, if you add a policy so that `weblogic` can reserve a connection, then all other users fail to reserve connections unless they are also explicitly added. The validation is done for WebLogic user credentials, not database user credentials.

You can protect JDBC resource operations by assigning Administrator methods which can limit the actions that an administrator may take upon a JDBC data source. These resources can be defined on the **Policies** tab on the **Security** tab associated with the data source. When you secure an individual data source, you can choose whether to protect `JDBC operations` using one or more of the following administrator methods:

- `admin`—The following methods on the `JDBCDataSourceRuntimeMBean` are invoked as `admin` operations: `clearStatementCache`, `suspend`, `forceSuspend`, `resume`, `shutdown`, `forceShutdown`, `start`, `getProperties`, and `poolExists`.
- `reserve`—Applications reserve a connection in the data source by looking up the data source and then calling `getConnection`. Giving a user the `reserve` permission enables

them to execute vendor-specific operations. Depending on the database vendor, some of these operations may have database security implications. See WebLogic Data Source Security Options.

- `shrink`—Shrinks the number of connections in the data source to the maximum of the currently reserved connections or to the initial size.
- `reset`—Resets the data source connections by shutting down and re-establishing all physical database connections. This also clears the statement cache for each connection. You can only reset data source connections that are running normally.
- `All`—An individual data source is protected by the union of the `Admin`, `reserve`, `shrink`, and `reset` administrator methods.

> **Note:**
>
> Be aware of the following:
>
> – If a security policy controls access to connections in a Multi Data Source, access checks are performed at both levels of the JDBC resource hierarchy (once at the Multi Data Source level, and again at the individual data source level). As with all types of WebLogic resources, this double-checking ensures that the most specific security policy controls access.

See Java DataBase Connectivity (JDBC) Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

The following table provides information on the user for permission checking when using the administrator method `reserve`:

Table 11-2 Determining the User when using the reserve Administration Method

API	Use-database-credential	User for permission checking
`getConnection()`	`True` or `False`	Current WebLogic user
`getConnection(`*user*, *password*`)`	`False`	User/password from API
`getConnection(`*user*, *password*`)`	`True`	Current WebLogic user

In summary, if a simple `getConnection()` is used or database credentials are enabled, the current user that is authenticated to the WebLogic system is checked. If database credentials are not enabled, then the user and password on the API are used. This feature is very useful to restrict what code and users can access your database.

For more information about securing server resources, see *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Data Source Security Example

Learn about the interactions and differences between Identity, Proxy, and Database Credentials with help of data source security example.

The following is an actual example of the interactions between `identity-based-connection-pooling-enabled`, `oracle-proxy-session`, and `use-database-credentials`.

On the database side, the following objects are configured:

- users: `scott`; `jdbcqa`; `jdbcqa3`
- `alter user jdbcqa3 grant connect through jdbcqa;`
- `alter user jdbcqa grant connect through jdbcqa;`

The following WebLogic users are configured:

- `weblogic`
- `wluser`

The following WebLogic data source objects are configured.

- Credential mapping `weblogic` to `scott`
- Credential mapping `wluser` to `jdbcqa3`
- Data Source configured with user `jdbcqa`
- All tests run with `Set Client ID` set to `true`.
- All tests run with `oracle-proxy-session` set to `false`.

The test program:

- Runs in servlet
- Authenticates to WebLogic as user `weblogic`

Table 11-3 Comparing Identity, Proxy, and Database Credentials

Use DB Credentials	Identity based	getConnection (scott,***)	getConnection (weblogic,***)	getConnection (jdbcqa3,***)	getConnection()
`true`	`true`	Identity `scott` Client `weblogic` Proxy `null`	`weblogic` fails – not a db user	User `jdbcqa3` Client `weblogic` Proxy `null`	Default `jdbcqa` Client `weblogic` Proxy `null`
`false`	`true`	`scott` fails – not a WebLogic user	User `scott` Client `scott` Proxy `null`	`jdbcqa3` fails – not a WebLogic user	User `scott` Client `scott` Proxy `null`
`true`	`false`	Proxy for `scott` failed	`weblogic` fails – not a db user	User `jdbcqa3` Client `weblogic` Proxy `jdbcqa`	Default `jdbcqa` Client `weblogic` Proxy `null`
`false`	`false`	`scott` fails – not a WebLogic user	User `jdbcqa` Client `scott` Proxy `null`	`jdbcqa3` fails – not a WebLogic user	Default `jdbcqa` Client `scott` Proxy `null`

If:

- `Set Client ID` is set to `false`, all cases would have `Client` set to `null`.
- The Oracle thin driver is not used, the one case with the non-`null` Proxy would throw an exception because proxy session is only supported with the Oracle thin driver.

When `oracle-proxy-session` is set to `true`, the only cases that pass (with a proxy of `jdbcqa`) are:

- Setting `use-database-credentials` to `true` and using `getConnection(jdbcqa3,…)` or `getConnection()`.
- Setting `use-database-credentials` is `false` and using `getConnection(wluser, …)` or `getConnection()`.

Using Encrypted Connection Properties

As part of a secure configuration, it may be necessary to provide one or more connection property values that should not appear as clear text in the connection properties of the data source descriptor file. These properties can be added using the `Encrypted Properties` attribute.

Note:

You cannot encrypt connection properties when creating a data source in the WebLogic Remote Console. It can only be done when updating an existing data source configuration.

- Best Practices
- WLST Examples

Best Practices

The following section provides information on best practices and tips when encrypting connection properties in the WebLogic Remote Console:

- When creating a data source:
  - Create it without the encrypted property and do not target the data source.
  - It may not be possible to test the connection without the encrypted property. You might want to temporarily test with a clear text property, then replace the clear text property with the encrypted property later.
  - Edit the data source by going to **Summary of JDBC Data Sources** page, select the **Data Source**, go to the **Configuration** tab and then select the **Connection Pool** tab.
- To enter values without clear text values displayed on the screen:
  - Save any other changes that you wish to make to this page and click the **Add Securely** button next to the **Encrypted Properties** text box.
  - On the **Add a new Encrypted Property** page, enter the property name and masked value, and click **OK**.
  - Repeat for additional encrypted property values.
  - Click **Save** when you have finished entering encrypted properties.
- You can enter several values at once if it is appropriate in your environment to display the encrypted values on the screen until the changes are saved.
  - List each `property=value` pair on a separate line in the **Encrypted Properties** field.

  - Click **Save** to encrypt the values.
- Activate your changes:
  - If the data source was untargeted: Go to the **Targets** tab, target the data source, and click **Save**.
  - If the data source was already active when the encrypted property values were added: Go to the **Targets** tab, untarget the data source, click **Save**, retarget the data source, and click **Save**.

WLST Examples

Providing WLST scripts examples to encrypt connection properties:

Use WLST to Update an Existing Data Source with Encrypted Properties

The following is an online WLST script that shows how to add an encrypted property to an existing data source named *genericds*:

```
connect('admin','password','t3://localhost:7001')
edit()
startEdit()
cd('JDBCSystemResources/genericds/JDBCResource/genericds/JDBCDriverParams/
genericds/Properties/genericds/Properties')
create('encryptedprop','Property')
cd('encryptedprop')
cmo. setEncryptedValueEncrypted(encrypt('foo'))
save()
activate()
```

Use WLST to Create Encrypted Properties

The following WLST script creates encrypted properties:

```
. . .
create('myProps','Properties')
cd('Properties/NO_NAME_0')
. . .
# Create other properties
. . .
p=create('javax.net.ssl.trustStoreType', 'Property')
p.setValue('JKS')

p=create('javax.net.ssl.trustStorePassword', 'Property')
p.setEncryptedValueEncrypted(encrypt('securityCommonTrustKeyStorePassPhrase'))
. . .
```

Using SSL and Encryption with Data Sources and Oracle Drivers

Use SSL to provide both data encryption and strong authentication for network connections to the database server. This topic provides additional information on using these features with WebLogic Server.

For more information, see JDBC Client-Side Security Features in the *Oracle® Database JDBC Developer's Guide*.

- Using SSL with Data Sources and Oracle Drivers
- Using Data Encryption with Data Sources and Oracle Drivers

Using SSL with Data Sources and Oracle Drivers

You must provide additional information on a variety of options that use SSL with data sources and Oracle drivers. The general requirement when using SSL, regardless of the option, is that you must specify a protocol of `tcps` in any URL.

For detailed information on configuring and using SSL with Oracle drivers, see:

- How-To Configure and Use Oracle JDBC Driver SSL with Oracle WebLogic Server
- http://www.oracle.com/technetwork/database/enterprise-edition/wp-oracle-jdbc-thin-ssl-130128.pdf.

If you use a provider that requires a password, such as the `javax.net.ssl.trustStorePassword` or `javax.net.ssl.keyStorePassword`, the value should be stored as an encrypted property. See Using Encrypted Connection Properties.

- Using SSL with Oracle Wallet
- Active GridLink ONS over SSL

Using SSL with Oracle Wallet

Oracle wallet can also be used with SSL. By using it correctly, passwords can be eliminated from the JDBC configuration and client/server configuration can be simplified by sharing the wallet). The following is a list of basic requirements to use SSL with Oracle wallet.

- Update the `sqlnet.ora` and `listener.ora` files with the location of the wallet. These files also indicate whether or not `SSL_CLIENT_AUTHENTICATION` is being used.
- If you use an auto-login wallet type, a password is not needed in the data source configuration to open the wallet. The store type for an auto-login wallet is *SSO* (not JKS or PKCS12) and the file name is `cwallet.sso`. If you use another provider type, use the encrypted property type to store the password as an encrypted value in the data source configuration.
- Enable the Oracle PKI provider in a WLS startup class using:

  ```
  Security.insertProviderAt(new oracle.security.pki.OraclePKIProvider (), 3);
  ```

- For encryption and server authentication, use the data source connection properties:

  ```
  javax.net.ssl.trustStore=location of wallet
  javax.net.ssl.trustStoreType="SSO"
  ```

- For client authentication, use the data source connection properties:

  ```
  javax.net.ssl.keyStore=location of wallet
  javax.net.ssl.keyStoreType="SSO"
  ```

- Wallets are created using the `orapki`. They need to be created based on the usage (encryption or authentication).

Common use cases are:

- *Encryption and server authentication*, which requires just a trust store.
- *Encryption and authentication of both tiers* (client and server), which requires a trust store and a key store.

Active GridLink ONS over SSL

You can use SSL to secure communication between an Active GridLink data source and the Oracle Notification Service (ONS) which is use to provide load balancing information and notification of node up/down events.

Use the following basic steps:

- Create an auto-login wallet and use the wallet on the client and server. The following is a sample sequence to create a test wallet for use with ONS.

  ```
  orapki wallet create -wallet ons -auto_login -pwd ONS_Wallet
  orapki wallet export -wallet ons -dn "CN=ons_test,C=US" -cert ons/cert.txt -pwd
  ONS_Wallet
  orapki wallet export -wallet ons -dn "CN=ons_test,C=US" -cert ons/cert.txt -pwd
  ONS_Wallet
  ```

- On the database server side:
  1. Define the wallet file directory in the file `$CRS_HOME/opmn/conf/ons.config`.
  2. Run `onsctl stop/start`.
- When configuring an Active GridLink data source, the connection to the ONS must be defined. In addition to the host and port, the wallet file directory must be specified. If you do not provide a password, a SSO wallet is assumed.

Using Data Encryption with Data Sources and Oracle Drivers

To use data encryption with the Oracle Thin driver, you must specify several connection properties, see Configuration Parameters in *Oracle® Database Advanced Security Administrator's Guide*. The following table maps the encryption and checksum configuration parameters to the string constants required when configuring data source descriptors using the Remote Console or WLST:

Table 11-4 Connection Encryption Parameters and WebLogic Configuration Constants

Client Configuration Parameter	WebLogic Server Configuration String Constant
OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL	oracle.net.encryption_client
OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES	oracle.net.encryption_types_client
OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL	oracle.net.crypto_checksum_client
OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES	oracle.net.crypto_checksum_types_client

12 Creating and Managing Oracle Wallet

Use Oracle wallet to store database credentials for WebLogic Server data source definition.

- What Is Oracle Wallet
- Where to Keep Your Wallet
- How to Create an External Password Store
- Define a WebLogic Server Data Source Using a Wallet
- Use a TNS Alias Instead of a DB Connection String
- Use DBClientData Modules for Portability

What Is Oracle Wallet

Oracle wallet provides an easy method to manage database credentials across multiple domains. It lets you update database credentials by updating the wallet instead of potentially changing many data source definitions. You accomplish updates by using a database connection string in the data source definition that is resolved by an entry in the wallet.

In addition, this feature can be made even easier by also using an Oracle TNS (Transparent Network Substrate) administrative file to hide the details of the database connection string (host name, port number, and service name) from the data source definition and instead use an alias. If the connection information changes, it is simply a matter of changing the `tnsnames.ora` file.

By using this approach, you remove the encrypted password from the data source descriptor so that it is portable across domains and the same wallet and `tnsnames.ora` file can be shared across multiple domains: that includes two different WebLogic Server domains or sharing credentials between WebLogic Server and the database. When used correctly, it makes having passwords in the data source configuration unnecessary.

Where to Keep Your Wallet

Oracle recommends that you create and manage the location of the wallet in a database environment. There, you'll have all the necessary commands and libraries, including the `$ORACLE_HOME/oracle_common/bin/mkstore` command. Often the storage of the wallet is managed by a database administrator and provided for use by the client. A configured wallet consists of two files, `cwallet.sso` and `ewallet.p12`, stored in a secure wallet directory.

> **Note:**
>
> Alternatively, you can install the Oracle Client Runtime package to provide the necessary commands and libraries to create and manage the wallet.

How to Create an External Password Store

Oracle wallet has an auto login feature that allows client access to the wallet contents without supplying a password. Use of this feature prevents exposing a clear text password on the client.

On the client, create a wallet by using the following syntax:

```
mkstore -wrl <wallet_location> -create
```

Where `wallet_location` is the path to the directory where you want to create and store the wallet.

This command creates a wallet with the auto login feature enabled at the location specified. Auto login lets the client access the wallet contents without supplying a password and prevents exposing a clear text password on the client.

The `mkstore` command prompts for a password that is used for subsequent commands. Passwords must have a minimum length of eight characters and contain alphabetic characters combined with numbers or special characters. For example:

```
mkstore -wrl /tmp/wallet -create
Enter password: mysecret
PKI-01002: Invalid password.
Enter password: mysecret1  (not echoed)
Enter password again: mysecret1 (not echoed)
```

> **Note:**
>
> Using a wallet moves the security vulnerability from a clear text password in the data source configuration file to an encrypted password in the wallet file. Make sure that the wallet file is stored in a secure location.

You can store multiple credentials for multiple databases in one client wallet. You *cannot* store multiple credentials (for logging in to multiple schemas) for the same database in the same wallet. If you have multiple login credentials for the same database, then they must be stored in separate wallets.

To add database login credentials to an existing client wallet, enter the following command:

```
mkstore -wrl <wallet_location> -createCredential <db_connect_string> <username>
<password>
```

Where:

- The `wallet_location` is the path to the directory where you created the wallet.
- The `db_connect_string` must be identical to the connection string that you specify in the URL used in the data source definition (the part of the string that follows the `@`). It can be either the short form or the long form of the URL. For example:

  `myhost:1521/myservice` OR

  ```
  (DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=myhost-scan)
  (PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=myservice)))
  ```

> **Note:**
>
> You should enclose this value in quotation marks to escape any special characters from the shell. Because this name is generally a long and complex value, an alternative is to use a TNS alias. See Use a TNS Alias instead of a DB Connect String.

- The `username` and `password` are the database login credentials.

Repeat this procedure for each database you want to use in a WebLogic data source.

For more information about using auto login and maintaining wallet passwords, see the *Oracle Database Advanced Security Administrator's Guide*.

Define a WebLogic Server Data Source Using a Wallet

To configure a WebLogic Server data source to use an Oracle wallet, copy the wallet files to a secure directory on the client machine and update the data source configuration files.

Use the following procedures to configure a WebLogic Server data source to use a wallet.

- Copy the Wallet Files
- Update the Data Source Configuration

Copy the Wallet Files

Copy the wallet files, `cwallet.sso` and `ewallet.p12`, from the database machine to the client machine and locate them in a secure directory.

Update the Data Source Configuration

Use the following steps to configure a WebLogic data source to use Oracle wallet:

1. Do not enter a user or password in the WebLogic Remote Console when creating a data source or deleting them from an existing data source. If a user, password, or encrypted password appears in the configuration, it overrides the Oracle wallet values.
2. Add the value `oracle.net.wallet_location=wallet_directory` to Connection Properties.

   Where *`wallet_directory`* is the secure directory location described in Copy the Wallet Files. An alternative method is use the `-Doracle.net.wallet_location` system property and add it to the `JAVA_OPTIONS`.

   NOTE: Oracle recommends using the connection property.

Use a TNS Alias Instead of a DB Connection String

Instead of specifying a matching database connection string in the URL and in the Oracle wallet, you can create an alias to map the URL information. The connection string information is stored in the `tnsnames.ora` file with an associated alias name. The alias name is then used both in the URL and the wallet.

Use the following steps to create a TNS alias:

1. Specify `oracle.net.tns_admin=tns_directory` to Connection Properties, where *tns_directory* is the directory location of the `tnsnames.ora` file. Alternatively, if all the data sources in the server have the same value, use the – `Doracle.net.tns_admin=<tns_directory>` with the location of a `tnsnames.ora` file.
2. Create or modify the `tnsnames.ora` file in the directory location specified by *tns_directory*. The entry has the form:

   ```
   alias=(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=host)(PORT=port))
   (CONNECT_DATA=(SERVICE_NAME=service)))
   ```

   Where *host* is the URL of a database listener, *port* is the port of a database listener, and *service* is the service name of the database to which you want to connect.

   For additional attributes that you can configure, see Local Naming Parameters (tnsnames.ora) in the *Database Net Services Reference*. Oracle recommends that the string be entered on a single line.
3. Use the alias in the data source definition URL by replacing the connection string with the alias. For example, change the **URL** attribute on the Connection Pool page in the Remote Console to `jdbc:oracle:thin:@alias`.

After the alias is created, it should not be necessary to modify the alias or the data source definition again. To change the user credential, update the wallet. To change the connection information, update the `tnsnames.ora` file. In either case, the data source must be redeployed. The simplest way to redeploy a data source is to untarget and target the data source in the WebLogic Remote Console. This configuration is supported for Oracle release 10.2 and later drivers.

Use DBClientData Modules for Portability

As of 14.1.2.0.0, you can use DBClientData modules, which enhance the integration of WLS on OCI and make it simple to connect to OCI Database services.

DBClientData modules provide a way to transfer the `tnsnames.ora` and Oracle wallet files, downloaded from OCI, to the machine or Kubernetes Pod where a WebLogic domain is running.

- What Are DBClientData Modules
- Why Use DBClientData Modules
- Manage DBClientData Modules
- Configure a Data Source to Use DBClientData Modules

What Are DBClientData Modules

DBClientData modules are `tnsnames.ora` files, wallet files, keystore and truststore files, basically all the database client connection data used by a data source, collocated in a new type of deployment module.

DBClientData modules are standalone deployment modules that are independent of the configuration of the data source instances that are going to use those data. A single DBClientData module can be referenced by multiple data sources and a single DBClientData module can be used by multiple domains.

Why Use DBClientData Modules

DBClientData modules make it easy to manage and move database client data, including the `tnsnames.ora` file and Oracle wallet files, into the file system of all servers in a domain.

In maximum availability architecture (MAA) environments (which provide disaster recovery and high availability in multi data center architectures) and in migration scenarios, it is important to include these files with the domain so that the domain can move; there is no requirement to make configuration changes even when the databases are different.

Using WebLogic deployment tools, you can deploy, distribute, undeploy, and redeploy DBClientData modules. Updates to DBClientData module files on the WebLogic Server Administration Server will be propagated to all servers in the domain. DBClientData module files reside under the `config` directory of the domain home directory, which ensures that the database client data goes with the domain when you use pack and unpack operations to move the domain.

Manage DBClientData Modules

You can manage DBClientData modules using existing WebLogic deployment tools, such as weblogic.Deployer, WLST online, REST APIs, WebLogic Remote Console, as well as WebLogic Deploy Tooling (WDT). The source of DBClientData modules can be a ZIP file or an exploded archive directory that contains the `tnsnames.ora` and wallet files.

Note:

To manage DBClientData modules, a new option, `dbClientdata`, has been added to the existing deploy, redeploy, distribute, and undeploy APIs. If specified for DBClientData modules, the `targets` parameter is *ignored* because they are always deployed, redeployed, distributed, and undeployed to all servers in a domain.

See the following sections for detailed information and examples of each management task using WLST. For weblogic.Deployer examples, see Deploying Oracle Wallet Files for JDBC Modules in *Deploying Applications to Oracle WebLogic Server*.

- Deploy DBClientData Modules
- Distribute DBClientData Modules
- Redeploy DBClientData Modules

Deploy DBClientData Modules

Deploying a DBClientData module performs the following actions.

- Uploads or copies the DBClientData files (for example, `tnsnames.ora` and Oracle wallet files) into a directory under `$DOMAIN_HOME/config/dbclientdata/<mydbclientdata>` on the Administration Server.
  - The source can be a ZIP file or an exploded archive directory.
  - If the source is a ZIP file, then it will be exploded after it is uploaded or copied.

- Generates a DBClientDataDirectoryMBean, which has its `sourcePath` attribute pointing to `$DOMAIN_HOME/config/dbclientdata/<mydbclientdata>`.
- Optionally, propagates the files to the same directory on all Managed Servers.
  - If Managed Servers are running when the deployment operation is performed, then the propagation happens right away.
  - If Managed Servers are down when the deployment operation is performed, then the propagation happens when the servers start.

Syntax:

```
deploy(name,path,[options])
```

The following example uses WLST to upload and deploy a DBClientData module in the default location, `$DOMAIN_HOME/config/dbclientdata`. As a result, `Users/joesmith/myWallet.zip` will be deployed to the `$DOMAIN_HOME/config/dbclientdata/myDBData` directory.

Example:

```
deploy('myDBData', '/Users/joesmith/myWallet.zip', upload='true', dbClientData='true');
```

Use the `-dbClientDataUploadPath` option to upload and deploy a DBClientData module in a custom location relative to the domain configuration directory.

The following example commands will deploy `/Users/joesmith/myWallet.zip` to the `$DOMAIN_HOME/config/myDBClientData/myDBData` directory and `$DOMAIN_HOME/config/myClientData/myDBClientData/myDBData` directory, respectively.

```
deploy('myDBData', '/Users/joesmith/myWallet.zip', upload='true',
dbClientDataUploadPath='myDBClientData', dbClientData='true');

deploy('myDBData', '/Users/joesmith/myWallet.zip', upload='true',
dbClientDataUploadPath='myClientData/myDBClientData', dbClientData='true');
```

Distribute DBClientData Modules

If the DBClientData module files already exist in the Administration Server's `dbClientDataDir`, then you can use any one of the WLS deployment tools to distribute those files to all Managed Servers in the domain.

- If Managed Servers are not running when the distribute operation is performed, then the propagation will happen when the servers start.
- If the `dbClientDataDir` for a deployment is not empty on the Managed Servers, then the existing contents are fully replaced by the new contents when the distribute operation is performed.

Syntax:

```
distributeApplication(path,[options])
```

The following example uses WLST to propagate a DBClientData module to Managed Servers when the files are already available in the expected directory on the Administration Server.

Example:

```
distributeApplication('/mydomain/config/dbclientdata/demoDBCD/myWallet.zip',
dbClientData='true');
```

The following example uses WLST to upload a DBClientData module to the Administration Server and propagate it to Managed Servers.

Example:

```
distributeApplication('/Users/joesmith/myWallet.zip', upload='true',
dbClientData='true');
```

Note:

For a DBClientData module, the`distributeApplication` WLST command derives the application name from the application path. In the two preceding examples, the derived application name is "demoDBCD" and "myWallet", respectively.

Redeploy DBClientData Modules

DBClientData modules are not versioned, therefore a redeploy operation is the same as an undeploy operation followed by a deploy operation. Upon redeploy, the content of the current `dbClientDataDir` on all servers is replaced by the content of the new DBClientData module source.

Syntax:

```
redeploy(name,[options])
```

The following example uses WLST to redeploy a DBClientData module.

Example:

```
redeploy('demoDBCD', dbClientData='true')
```

Use the `-dbClientDataUploadPath` option to upload and redeploy a DBClientData module in a custom location under the domain configuration directory.

```
redeploy('demoDBCD', upload='true', dbClientDataUploadPath='myDBClientData',
dbClientData='true');
```

Configure a Data Source to Use DBClientData Modules

After you Deploy DBClientData Modules, configure a WebLogic Server data source to use DBClientData modules using the same steps as described in the following sections:

- Update the Data Source Configuration
- Use a TNS Alias Instead of a DB Connection String

Note:

You need to point to the `dbClientDataDir` where the corresponding `tnsnames.ora` and wallet files are located.

- The default file system location is`$DOMAIN_HOME/config/dbclientdata/<dbclientDatamodulename>`.
- However, if the DBClientData module is included in a domain that was created or updated using WebLogic Deploy Tooling, then its default location will be different:
  - `$DOMAIN_HOME/config/wlsdeploy/dbWallets/<dbclientmodulename>`
  - **NOTE**: The WDT default location may change. For detailed information, see Archive File in the WebLogic Deploy Tooling documentation.
- Remember that if any of the data in a DBClientData module or data source configuration has been changed, then a data source must be restarted for the updated data to be used.

13 Deploying Data Sources on Servers and Clusters

Learn how to deploy data sources on servers and clusters.

- Deploying Data Sources on Servers and Clusters
- Minimizing Server Startup Hang Caused By an Unresponsive Database
  To minimize the chances of the server hanging during start-up, use the `JDBCLoginTimeoutSeconds` attribute on the ServerMBean.

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.

Minimizing Server Startup Hang Caused By an Unresponsive Database

To minimize the chances of the server hanging during start-up, use the `JDBCLoginTimeoutSeconds` attribute on the ServerMBean.

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.

An alternative is to set the `Initial Capacity` for the data source to 0. That means that no connections are created when the data source is deployed and the database need not even be available at that time. Connection creation is deferred until the application needs them.

14 Using WebLogic Server with Oracle RAC

Oracle WebLogic Server provides strong support for Oracle Real Application Clusters (RAC), minimizing database access time while allowing transparent access to rich pooling management functions that maximizes both connection performance and availability.
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.

This chapter describes the requirements and configuration tasks for using Oracle Real Application Clusters (Oracle RAC) with WebLogic Server.

- Overview of Oracle Real Application Clusters
  Oracle 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.
- Software Requirements
  Learn about the software requirements for using WebLogic Server with Oracle RAC.
- JDBC Driver Requirements
  To use WebLogic Server with Oracle RAC, your WebLogic JDBC data sources must use the Oracle JDBC Thin driver 11g or later to create database connections.
- Hardware Requirements
  A typical WebLogic Server/Oracle RAC configuration includes a WebLogic Server cluster, an Oracle RAC cluster, and hardware for shared storage.
- Configuration Options in WebLogic Server with Oracle RAC
  When using WebLogic Server with Oracle RAC, configure the WebLogic domain so that it interacts with Oracle RAC instances.

Overview of Oracle Real Application Clusters

Oracle 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.

Oracle 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 hosts 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.

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 Oracle 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 Oracle RAC nodes, storage, or both.

Software Requirements

Learn about the software requirements for using WebLogic Server with Oracle RAC.

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

- Operating system patches required to support Oracle RAC. See the release notes from Oracle for details.
- Oracle database management system. See Oracle® Fusion Middleware Licensing Information.
- Clustering software for your operating system. See the Oracle documentation for supported clustering software and cluster configurations.
- Shared storage software, such as Oracle Automatic Storage Management (ASM). Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

> **Note:**
>
> See Supported Configurations in *What's New in Oracle WebLogic Server* 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.

JDBC Driver Requirements

To use WebLogic Server with Oracle RAC, your WebLogic JDBC data sources must use the Oracle JDBC Thin driver 11g or later to create database connections.

Hardware Requirements

A typical WebLogic Server/Oracle RAC configuration 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 *Administering Clusters for Oracle WebLogic Server* 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 Oracle 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.

Configuration Options in WebLogic Server with Oracle RAC

When using WebLogic Server with Oracle RAC, configure the WebLogic domain so that it interacts with Oracle RAC instances.

- Choosing a WebLogic Server Configuration for Use with Oracle RAC
- Validating Connections when using WebLogic Server with Oracle RAC
- Additional Considerations When Using WebLogic Server with Oracle RAC

Choosing a WebLogic Server Configuration for Use with Oracle RAC

When using WebLogic Server with Oracle RAC, you can configure WebLogic Data Sources by considering the following alternatives:

- Using Active GridLink (AGL) data sources, see Oracle® Fusion Middleware Licensing Information. AGL offers the best integration to Oracle RAC by providing the best performance and high availability. AGL supports Fast Connection Failover, automatic and transparent addition, and removal of RAC instances. It also automatically handles when nodes go down and come up without waiting for connection failures and successes. AGL affirms runtime connection load balancing (RCLB) providing the best performance as the database drives load balancing of connections through the AGL data source, independent of the database topology See Using Active GridLink Data Sources.
- Multi Data Source (MDS) provides failover and load balancing capabilities across the instances of the Real Application Clusters (RAC). MDS failover is handled at the MDS level when an Oracle RAC instance becomes unavailable. MDS load balancing follows a Round Robin pattern across the RAC instances. See Configuring Connections to Services on Oracle RAC Nodes
- To connect to multiple Oracle RAC instances when using Global transactions (XA), Oracle recommends the use of transaction-aware AGL or MDS, which support failover and load balancing, to connect to the Oracle RAC nodes.
  - For configuring AGL with Global transactions see Configure Transaction Options and GridLink Affinity Policies - XA Affinity
  - For configuring MDS with Global transactions see Using Multi Data Sources with Global Transactions.
- To connect to multiple Oracle RAC instances when not using XA, Oracle recommends the use of non-transaction-aware AGL or MDS to connect to the Oracle RAC nodes.
  - For configuring AGL without Global transactions see Configure Transaction Options
  - For configuring MDS without Global transactions see Using Multi Data Sources without Global Transactions .

Using a Generic data source for XA with Oracle RAC is not supported. See Generic Data Source Handling for Oracle RAC Outages.

Validating Connections when using WebLogic Server with Oracle RAC

Applications can use the JDBC 4.0 Connection.isValid API to verify connection viability.

WebLogic Server does not support `oracle.ucp.jdbc.ValidConnection.isValid` or `oracle.ucp.jdbc.ValidConnection.setInvalid`.

Additional Considerations When Using WebLogic Server with Oracle RAC

The Distributed Transaction Processing (DTP) attribute on a database service should not be used to coordinate transactions when using Active GridLink or Multi Data Sources with Oracle RAC. This option implies that the service is guaranteed to run on only one RAC instance at any time. Transaction affinity to a single instance is automatically managed by WebLogic Server for either Active GridLink or Multi Data Sources. This allows the whole RAC cluster to be available for distributed transactions, as opposed to DTP limiting all transactions for the service to a single RAC instance.

15

Using JDBC Drivers with WebLogic Server

WebLogic Server uses JDBC drivers to provide access to various databases. WebLogic Server comes with a default set of JDBC drivers but third-party JDBC drivers can also be used.

- JDBC Driver Support
  WebLogic Server provides support for application data access to any database using a JDBC-compliant driver.
- JDBC Drivers Installed with WebLogic Server
  The Oracle JDBC Thin driver 23ai is installed with Oracle WebLogic Server 14.1.2.0.0. In addition to the Oracle Thin Driver, the mySQL Connector/J 8.0 (`mysql-connector-j-8.2.0.jar`) JDBC driver, WebLogic-branded DataDirect drivers are also installed with WebLogic Server.
- Adding Third-Party JDBC Drivers Not Installed with WebLogic Server
- Globalization Support for the Oracle Thin Driver

JDBC Driver Support

WebLogic Server provides support for application data access to any database using a JDBC-compliant driver.

The JDBC-compliant driver needs to meet the following requirements:

- The driver must be thread-safe.
- The driver must implement standard JDBC transactional calls, such as `setAutoCommit()` and `setTransactionIsolation()`, when used in transactional aware environments.
- If the driver that does not implement serializable or remote interfaces, it cannot pass objects to an RMI client application.

When WebLogic Server features use a database for internal data storage, database support is more restrictive than for application data access. The following WebLogic Server features require internal data storage:

- Container Managed Persistence (CMP)
- Rowsets
- JMS/JDBC Persistence and use of a WebLogic JDBC Store
- JDBC Session Persistence
- RDBMS Security Providers
- Database Leasing (for singleton services and server migration)
- JTA Logging Last Resource (LLR) optimization.

JDBC Drivers Installed with WebLogic Server

The Oracle JDBC Thin driver 23ai is installed with Oracle WebLogic Server 14.1.2.0.0. In addition to the Oracle Thin Driver, the mySQL Connector/J 8.0 (`mysql-connector-`

j-8.2.0.jar) JDBC driver, WebLogic-branded DataDirect drivers are also installed with WebLogic Server.

The drivers files are named ojdbc11.jar for JDK17 and JDK21.

Note:

See Using WebLogic-branded DataDirect Drivers in *Developing JDBC Applications for Oracle WebLogic Server*.

These drivers are installed in subdirectories of $ORACLE_HOME/oracle_common/modules. The manifest in the weblogic.jar lists this file so that it is loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add this JDBC driver 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 Configurations in *What's New in Oracle WebLogic Server* .

Note:

WebLogic Server includes a version of the Derby DBMS installed with the WebLogic Server examples in the WL_HOME\common\derby directory. Derby is an all-Java DBMS product included in the WebLogic Server distribution solely in support of demonstrating the WebLogic Server examples. For more information about Derby, see http://db.apache.org/derby.

Adding Third-Party JDBC Drivers Not Installed with WebLogic Server

To use third-party JDBC drivers that are not installed with WebLogic Server, you can add them to the DOMAIN_HOME/lib directory.Here, DOMAIN_HOME represents the directory in which the WebLogic Server domain is configured. The default path is ORACLE_HOME/user_projects/domains.
For more information, see Adding JARs to the Domain /lib Directory in *Developing Applications for Oracle WebLogic Server*.

Note:

In previous releases, adding a new JDBC driver or updating a JDBC driver where the replacement JAR has a different name than the original JAR required updating the WebLogic Server's classpath to include the location of the JDBC driver classes. This is no longer required.

Using a Third-Party JAR File in DOMAIN_HOME/lib

Using a third-party JAR file in `DOMAIN_HOME/lib` is only supported for third-party JDBC drivers that are not installed with WebLogic Server. The drivers installed with WebLogic Server are described in JDBC Drivers Installed with WebLogic Server.

When you use a third-party JAR file in the `DOMAIN_HOME/lib` directory, note the following:

- The classloader that gets created is a child of the system classpath classloader in WebLogic Server.
- Any classes that are in JARs in this directory are visible only to Jakarta EE applications in the server, such as EAR files.
- You can use the WebLogic Remote Console and WLST online to configure and manage the JAR files. (You may also be able to use WLST offline because the data source is not deployed.)
- These JAR files do not work when run from a standalone client (such as the t3 RMI client) or standalone applications (such as `java utils.Schema`).
- If there are multiple domain directories involved (that is, multiple machines without a shared file system), the JAR file must be installed in `/lib` in each domain directory.
- WebLogic Server use of methods called on third-party drivers (such as TimesTen `abort` and `DB2 setDB2ClientUser`) is supported.

> **Note:**
>
> For details on WebLogic Server functionality supported with these JAR files, see Database Interoperability in *What's New in Oracle WebLogic Server*, and the appropriate version of the Oracle Fusion Middleware Supported System Configurations matrix documentation for specific database driver and DB version certification information.

Data Source Support

Third-party JAR files installed `in /lib` can be used with the following:

- All data source types supported by WebLogic Server system resources including Generic, Multi Data Source, and Active GridLink. The Universal Connection Pool data source does not apply since the UCP JAR is not third-party.
- Packaged data sources in an EAR or a WAR.
- Jakarta EE 8 data source definition defined in an EAR or WAR.

Although not JDBC methods, using a third-party JAR file `in /lib` does apply to WebLogic Server data source callbacks like Multi Data Source failover, connection, replay, and harvesting.

Example 15-1 Example of Using a Third-Party JAR File in /lib

The following example shows the files contained in a standalone WAR file named `getversion.war`. The Derby JAR files are located in WEB-`INF/lib` or `DOMAIN_HOME/lib` (or both). The class file is compiled and installed at `WEB-INF/classes/demo/GetVersion.class`.

```
<web-app>
  <welcome-file-list>
    <welcome-file>welcome.jsp</welcome-file>
  </welcome-file-list>
  <display-name>GetVersion</display-name>
  <servlet>
    <description></description>
    <display-name>GetVersion</display-name>
    <servlet-name>GetVersion</servlet-name>
    <servlet-class>
       demo.GetVersion
    </servlet-class>
  </servlet>
<!-- Data source description can go in the web.xml descriptor or as an
annotation in the java code - see below
  <data-source>
    <name>java:global/DSD</name>
    <class-name>org.apache.derby.jdbc.ClientDataSource</class-name>
    <port-number>1527</port-number>
    <server-name>localhost</server-name>
    <database-name>examples</database-name>
    <transactional>false</transactional>
  </data-source>
-->
</web-app>

WEB-INF/weblogic.xml

<weblogic-web-app>
  <container-descriptor>
    <prefer-web-inf-classes>true</prefer-web-inf-classes>
  </container-descriptor>
</weblogic-web-app>

Java file

package demo;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.SQLException;
import javax.annotation.Resource;
import javax.annotation.sql.DataSourceDefinition;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
```

```
@DataSourceDefinition(name="java:global/DSD",
className="org.apache.derby.jdbc.ClientDataSource",
portNumber=1527,
serverName="localhost",
databaseName="examples",
transactional=false
)
@WebServlet(urlPatterns = "/GetVersion")
public class GetVersion extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {
  @Resource(lookup = "java:global/DSD")
  private DataSource ds;

  public GetVersion() {
    super();
  }

  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    doPost(request, response);
  }

  protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");

    PrintWriter writer = response.getWriter();
    writer.println("<html>");
    writer.println("<head><title>GetVersion</title></head>");
    writer.println("<body>" + doit() +"</body>");
    writer.println("</html>");
    writer.close();
  }

  private String doit() {
    String ret = "FAILED";
    Connection conn = null;
    try {
      conn = ds.getConnection();
      ret = "Connection obtained with version= " +
        conn.getMetaData().getDriverVersion();
    } catch(Exception e) {
      e.printStackTrace();
    } finally {
      try {
        if (conn != null)
          conn.close();
      } catch (Exception ignore) {}
    }
    return ret;
  }
}
```

Globalization Support for the Oracle Thin Driver

For globalization support with the Oracle Thin driver, Oracle supplies the `orai18n.jar` file.This file 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` and `orai18n-mapping.jar` in your `CLASSPATH`.

The `orai18n.jar` and `orai18n-mapping.jar` are included with the WebLogic Server installation in the *`ORACLE_HOME`*`\oracle_common\modules\oracle.nlsrtl` folder. These files are *not* referenced by the `weblogic.jar manifest` file, so you must add them to your `CLASSPATH` before they can be used.

16

Monitoring WebLogic JDBC Resources

For monitoring WebLogic JDBC resources you need to understand how 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 run-time 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 run-time statistics, profile information over a period of time, logging, and debugging to help you keep your WebLogic domain running smoothly.
You can use the run-time 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.

- Viewing Run-Time Statistics
- Profile Logging
  WebLogic Server uses a data source profile log to store events.
- Collecting Profile Information
- Debugging JDBC Data Sources
  Once you narrow the problem down to a specific application, you can activate the WebLogic Server debugging features to isolate the problem with the application.

Viewing Run-Time Statistics

Viewing run-time statistics allows you to monitor the data sources in your WebLogic domain.

- Data Source Statistics
- Prepared Statement Cache Statistics

Data Source Statistics

You can view run-time statistics for a data source using the WebLogic Remote Console (see *Monitor Statistics for JDBC Data Sources*) 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 *MBean Reference for Oracle WebLogic Server*.

Prepared Statement Cache Statistics

You can view run-time statistics for a prepared statement cache via the *Oracle WebLogic Remote Console Online Help* or through the `JBCDataSourceRuntimeMBean`. For more information, see JDBCDataSourceRuntimeMBean in the *MBean Reference for Oracle WebLogic Server*.

Profile Logging

WebLogic Server uses a data source profile log to store events.

The profile log has the following benefits:

- Log-rotation—provides the ability to configure, rotate, and retire old data using the standard WebLogic logging implementation. See the DataSourceLogFileMBean in *MBean Reference for Oracle WebLogic Server*.
- Data accessibility—provides the ability to use common text editors, the WLDF Data Accessor, or the WebLogic Remote Console. See Accessing Diagnostic Data.

Basic characteristics of the log for data source profiling are:

- A single log file is used for all data source profile types. Each profile record has the profile type name for filtering. See Profile Types.
- A single log file is used for all data sources on the server. Each profile record has the decorated data source name for filtering (fully qualified with `application@module@component`, if applicable). See the DataSourceLogFileMBean in *MBean Reference for Oracle WebLogic Server*.

For more information on WebLogic logging services, see Understanding WebLogic Logging Services in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

Collecting Profile Information

If the statistics indicate a problem in your WebLogic 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 profile log.
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.

The fields contain different information for different profile types:

- Profile Types
- Accessing Diagnostic Data
- Callbacks for Monitoring Driver-Level Statistics

Profile Types

For each of the profile types in this section, the `User` information provides a stack trace of the thread that allocated the connection and is associated with the operation being profiled. By default, the value is not set because of the overhead in tracking this information. To obtain this information, you must also enable profiling of connection leaks in addition to profile type that you want to track. For more information about profiling connection leaks, see Connection Leak (WEBLOGIC.JDBC.CONN.LEAK).

You can choose to profile the following information about data sources and the prepared statement cache:

- Connection Usage (WEBLOGIC.JDBC.CONN.USAGE)
- Connection Reservation Wait (WEBLOGIC.JDBC.CONN.RESV.WAIT)

- Connection Reservation Failed (WEBLOGIC.JDBC.CONN.RESV.FAIL)
- Connection Leak (WEBLOGIC.JDBC.CONN.LEAK)
- Connection Last Usage (WEBLOGIC.JDBC.CONN.LAST_USAGE)
- Connection Multithreaded Usage (WEBLOGIC.JDBC.CONN.MT_USAGE)
- Statement Cache Entry (WEBLOGIC.JDBC.STMT_CACHE.ENTRY)
- Statements Usage (WEBLOGIC.JDBC.STMT.USAGE)
- Connection Unwrap (WEBLOGIC.JDBC.CONN.UNWRAP)
- JDBC Object Closed Usage (WEBLOGIC.JDBC.CLOSED_USAGE)
- Local Transaction Connection Leak (WEBLOGIC.JDBC.CONN.LOCALTX_LEAK)
- Example Profile Information Record Log

Connection Usage (WEBLOGIC.JDBC.CONN.USAGE)

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.

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 (WEBLOGIC.JDBC.CONN.RESV.WAIT)

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 (WEBLOGIC.JDBC.CONN.RESV.FAIL)

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 (WEBLOGIC.JDBC.CONN.LEAK)

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. Connection leak profiling must be enabled to get user stack trace information for any of the profile types.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

To specify the length of time before a reserved connection is considered leaked, do one of the following:

- Set `Inactive Connection Timeout Seconds` to a value greater than zero. WebLogic prints a stack trace of where a JDBC pool connection was reserved. The stack trace is printed after the `Inactive Connection Timeout Seconds` expires.
- Set `Connection Leak Timeout Seconds` to a value greater than zero. The value specifies the number of seconds that a JDBC connection needs to be held by an application before triggering a connection leak diagnostic profiling record. If set to 0, the timeout is disabled.

Connection Last Usage (WEBLOGIC.JDBC.CONN.LAST_USAGE)

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 (WEBLOGIC.JDBC.CONN.MT_USAGE)

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 (WEBLOGIC.JDBC.STMT_CACHE.ENTRY)

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 (WEBLOGIC.JDBC.STMT.USAGE)

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

Connection Unwrap (WEBLOGIC.JDBC.CONN.UNWRAP)

Enable connection unwrap profiling to collect profile information about application components that access the underlying JDBC connection using either the `getVendorObject` WebLogic extension API or the JDBC 4.0 method `unwrap`. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - stack trace of where the object was unwrapped
- User - stack trace of the thread unwrapping the object
- Timestamp - time stamp showing when the object was unwrapped.

JDBC Object Closed Usage (WEBLOGIC.JDBC.CLOSED_USAGE)

Enable JDBC object usage profiling to collect profile information about JDBC objects (Connection, Statement, or ResultSet) that are accessed after the close() method has been invoked. This information can help you determine both the thread that initially closed the object and the thread that attempted to access the closed object. The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - stack trace of the current thread attempting to close the object
- User - stack trace of the thread that closed the object plus where the close was done
- Timestamp - time stamp showing when the object was closed

Local Transaction Connection Leak (WEBLOGIC.JDBC.CONN.LOCALTX_LEAK)

Enable JDBC local transaction connection leak profiling to collect profile information about application components that leak a local transaction (start it but don't commit or rollback the transaction). The log record will include the call stack and details about the thread releasing the connection.The record contains the following information:

- PoolName - name of the data source to which this connection belongs
- ID - stack trace of the thread that is releasing the connection
- User - stack trace of the reserving thread plus a stack trace of the thread at the time the connection was closed
- Timestamp - time stamp showing when the connection was closed

Example Profile Information Record Log

The following is an example profile information record for Statements Usage (WEBLOGIC.JDBC.STMT.USAGE) from a standard output log.

```
####<JDBC Data Source-0> <WEBLOGIC.JDBC.STMT.USAGE> <0> <java.lang.Exception
     at
.
.
.

weblogic.servlet.provider.ContainerSupportProviderImpl$WlsRequestExecutor.run( ContainerS
upportProviderImpl.java:254)
     at weblogic.work.ExecuteThread.execute(ExecuteThread.java:295)
     at weblogic.work.ExecuteThread.run(ExecuteThread.java:254)
 > <select 1 from dual>
```

Each component of the profile log is surrounded by brackets ("<" and ">"):

- The PoolName—`JDBC Data Source-0`
- The Profile Type— `WEBLOGIC.JDBC.STMT.USAGE`
- The Timestamp—`0` (milliseconds)
- User—`java.lang.Exception at . . . at weblogic.work.ExecuteThread.run(ExecuteThread.java:254`
- ID—`select 1 from dual`

Accessing Diagnostic Data

You can use one of the following methods to access diagnostic data:

- The WebLogic Remote Console.
- The Data Accessor component of the WebLogic Diagnostic Framework (WLDF). See Accessing Diagnostic Data With the Data Accessor in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*
- Manually review information using text editors.
- When running with `DataSource` profiling, the default harvesting time is 300 seconds so you may not be able to view data immediately. You may need to set the harvest time to a small

value (say 5 seconds) to better visualize results. To see all connections, take a diagnostic image. To see the stack trace, enable leak profiling.

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 narrow the problem down to a specific application, you can activate the WebLogic Server debugging features to isolate the problem with the application.

- Enabling Debugging
- JDBC Debugging Scopes
- Set Debugging for UCP or ONS
- Request Dyeing

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
- Enable Debugging Using the WebLogic Remote Console
  To track down problems within the application you can enable debugging using the WebLogic Remote Console.
- Enable Debugging Using the WebLogic Scripting Tool
- Changes to the config.xml File

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 Remote Console

To track down problems within the application you can enable debugging using the WebLogic Remote Console.

To enable debugging set the following values:

1. In WebLogic Remote Console, click **Edit Tree**.
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. Select the check box for the debug scopes or attributes you want to modify.
6. Click **Save**.
7. To activate these changes, select **Shopping Cart** and click **Commit Changes**.

   Not all changes take effect immediately—some require a restart.

   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 Example 16-1:

Example 16-1 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

The following are 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.
- `DebugJDBCONS (scope weblogic.jdbc.rac)` - trace low-level ONS debugging.
- `DebugJDBCRAC (scope weblogic.jdbc.rac)` - trace RAC debugging.
- `DebugJDBCUCP (scope weblogic.jdbc.rac)` - trace low-level UCP debugging.
- `DebugJDBCReplay (scope weblogic.jdbc.rac)` - trace Replay debugging.
- `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:

To get driver level tracing for Oracle, you need to use `ojdbc6_g.jar` instead of `ojdbc6.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).

- `DebugJTAJDBC (scope weblogic.jdbc.transaction)` - trace transaction debugging.

Set Debugging for UCP or ONS

Debugging UCP

Set UCP debugging directly using:

```
oracle.ucp.level = FINEST;
oracle.ucp.jdbc.PoolDataSource = WARNING;
```

Debugging ONS

To enable debugging for ONS, you must configure `java.util.logging`.

```
-Djava.util.logging.config.file=configfile
-Doracle.ons.debug=true
```

In this command, `configfile` is the path and file name of the configuration property file property used by standard JDK logging to control the log output format and logging level. The `configfile` must include the following line:

```
oracle.ons.level=FINEST
```

For more information, see java.util.logging in *Java Platform Standard Edition API Specification*.

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 Diagnostics Framework for Oracle WebLogic Server*.

17 Managing WebLogic JDBC Resources

Learn how to use the WebLogic Remote Console, command line, JMX programs, or WebLogic Scripting Tool (WLST) scripts to manage the JDBC data sources in your domain.

- Testing Data Sources and Database Connections
  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.
- Managing the Statement Cache for a Data Source
  WebLogic Server creates a statement cache of each connection in a data source. When a prepared statement or callable statement is used on a connection, WebLogic Server caches the statement so that it can be reused.
- Shrinking a Connection Pool
  Use the Shrink option to drop some connections from the data source when a peak usage period has ended. This option frees up WebLogic Server and DBMS resources.
- Resetting a Connection Pool
  Use the Reset option to close and recreate all available database connections in a data source.
- Suspending a Connection Pool
  Use the `suspend()` and `forceSuspend()` options to suspend a data source.
- Resuming a Connection Pool
  Use the Resume option to re-use a suspended data source.
- Shutting Down a Data Source
  Use the Shutdown and Force Shutdown options to shut down a data source.
- Starting a Data Source
  Use the Start option to start a data source which has been shut down.
- Managing DBMS Network Failures

Testing Data Sources and Database Connections

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 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.

To manually test a connection from a data source, you can use the Test Data Source feature in the WebLogic Remote Console (see Test JDBC Data Sources) or the `testPool()` method in the `JDBCDataSourceRuntimeMBean`.

```
JDBCDataSourceRuntimeMBean.testPool
```

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 WebLogic Remote 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

WebLogic Server creates a statement cache of each connection in a data source. 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.

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 WebLogic Remote Console or you can programmatically clear the statement cache for an *individual* connection.

Note:

When the JDBC 4.0 `setPoolable(false)` method is called for a WebLogic data source that has prepared statement caching enabled, the statement is removed from the cache in addition to calling the method on the driver object.

- Clearing the Statement Cache for a Data Source
- Clearing the Statement Cache for a Single Connection

Clearing the Statement Cache for a Data Source

You can manually clear the statement cache for all connections in a data source using the WebLogic Remote Console or with the `clearStatementCache()` method on the `JDBCDataSourceRuntimeMBean`.

```
JDBCDataSourceRuntimeMBean.clearStatementCache
```

For more information, see *Oracle WebLogic Remote Console Online Help*.

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 Connection Pool

Use the Shrink option to drop some connections from the data source when a peak usage period has ended. This option frees up WebLogic Server and DBMS resources.

A data source has a set of properties that define the initial, minimum, and maximum number of connections in the pool (`initialCapacity`, `minCapacity`, and `maxCapacity`). A data source automatically adds one connection to the pool when all connections are in use. When the pool reaches `maxCapacity`, 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 in the WebLogic Remote Console (see *Oracle WebLogic Remote Console Online Help*) or the `shrink()` method on the `JDBCDataSourceRuntimeMBean`.

JDBCDataSourceRuntimeMBean.shrink

When you shrink a data source, WebLogic Server reduces the number of connections in the pool to the greater of either the `minCapacity` or the number of connections currently in use. The pool is decreased gradually to minimize thrashing. The number of unused connections is cut in half each time automatic shrinking is performed.

Resetting a Connection Pool

Use the Reset option to close and recreate all available database connections in a data source.

Reset option is available in the WebLogic Remote Console (see *Oracle WebLogic Remote Console Online Help*) or the `reset()` method on the `JDBCDataSourceRuntimeMBean`.

JDBCDataSourceRuntimeMBean.reset

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 Connection Pool

Use the `suspend()` and `forceSuspend()` options to suspend a data source.

The Suspend and Force Suspend options in the WebLogic Remote Console (see *Oracle WebLogic Remote Console Online Help*) or the `suspend()` and `forceSuspend()` methods in the `JDBCDataSourceRuntimeMBean`.

JDBCDataSourceRuntimeMBean.suspend
JDBCDataSourceRuntimeMBean.forceSuspend

When you suspend a data source (not forcibly suspend), the data source is marked as disabled and applications cannot reserve 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 gracefully suspend a data source, the following occurs:

- The data source is immediately marked as suspended at the beginning of the operation and no further connections are created on the data source.
- Idle (not reserved) connections are marked as disabled.
- After a timeout period for the suspend operation, all remaining connections in the pool are marked as suspended and the following exception is thrown for any operations on the connection, indicating that the data source is suspended:

  ```
  java.sql.SQLRecoverableException: Connection has been administratively
  disabled. Try later.
  ```
- If graceful suspend is done as part of a graceful shutdown operation, connections are immediately closed when no longer reserved or at the end of the timeout period. If not done as part of a shutdown operation, these connections remain in the pool and are not closed because the pool may be resumed.

A graceful suspend can be done synchronously or asynchronously.

The synchronous operation does not have a timeout period on the method. By default, the timeout period is 60 seconds. You can change the value of this timeout period by configuring or dynamically setting Inactive Connection Timeout Seconds to a non-zero value. There is no upper limit on the inactive timeout period. Note that the processing actually checks for in-use (reserved) resources every tenth of a second so if the timeout value is set to 2 hours and all reserved resources are released a second later, the shutdown will complete a second later.

The asynchronous operation takes a timeout value in seconds. It returns a `JDBCDataSourceTaskRuntimeMBean` that can be used to check the status of the operation. The `getProgress()` method returns `TaskRuntimeMBean.PROGRESS_SUCCESS ("success")`, `TaskRuntimeMBean.PROGRESS_FAILED ("failed")`, or `TaskRuntimeMBean.PROGRESS_PROCESSING ("processing")`. The `getStatus()` method returns `"SUCCESS"`, `"FAILURE"`, and now `"PROCESSING"`. There can be multiple task MBeans in existence. The next operation call on the data source will clean up MBeans for tasks that have been completed for at least 30 minutes. Note that once a suspend or shutdown operation is started, the other operations will fail immediately but a task MBean is still created. The `isRunning()` method returns true if suspend or shutdown is still running. Timeout of the operation is controlled by the timeout parameter on the new task operations. If set to 0, the default is used. The default is to use `Inactive Connection Timeout Seconds` if set or 60

seconds. If you want a minimal timeout, set the value to 1. If you want no timeout, set it to a large value (not recommended).

When you forcibly suspend a data source, all pool connections are destroyed and any subsequent attempt to use reserved connections fail. Any transactions on the connections that are closed are rolled back.

Resuming a Connection Pool

Use the Resume option to re-use a suspended data source.

The Resume option is available in the WebLogic Remote Console (see *Oracle WebLogic Remote Console Online Help*) or the `resume()` method on the `JDBCDataSourceRuntimeMBean`.

JDBCDataSourceRuntimeMBean.resume

When you resume a data source, WebLogic Server marks the data source as enabled and allows applications to reserve 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 exactly where they left off. If you forcibly suspended the data source, clients will have to reserve new connections to proceed.

> **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

Use the Shutdown and Force Shutdown options to shut down a data source.

The Shutdown and Force Shutdown options are available in the WebLogic Remote Console (see *Oracle WebLogic Remote Console Online Help*) or the `shutdown()` and `forceShutdown()` methods in the `JDBCDataSourceRuntimeMBean`.

JDBCDataSourceRuntimeMBean.shutdown
JDBCDataSourceRuntimeMBean.forceShutdown

A graceful (non-forced) data source shutdown operation involves first gracefully suspending the data source and then releasing the associated resources including the connections. See the description above for details of gracefully suspending the data source. After the data source is gracefully suspended, all remaining in-use connections are closed and the data source is marked as shut down.

A graceful shutdown can be done synchronously or asynchronously.

The synchronous operation does not have a timeout period on the method. The timeout period is 60 seconds by default. This can be changed by configuring or dynamically setting **Inactive Connection Timeout Seconds** to a non-zero value (note that this value is overloaded with another feature when connection leak profiling is enabled). There is no upper limit on the inactive timeout. Note that the processing actually checks for in-use (reserved) resources every tenth of a second so if the timeout value is set to 2 hours and it's done a second later, it will complete a second later.

The asynchronous operation takes a timeout value in seconds. It returns a `JDBCDataSourceTaskRuntimeMBean` that can be used to check the status of the operation. The `getProgress()` method returns `TaskRuntimeMBean.PROGRESS_SUCCESS ("success")`, `TaskRuntimeMBean.PROGRESS_FAILED ("failed")`, or `TaskRuntimeMBean.PROGRESS_PROCESSING ("processing")`. The `getStatus()` method returns `"SUCCESS", "FAILURE", and now "PROCESSING"`. There can be multiple task MBeans in existence. The next operation call on the data source will clean up MBeans for tasks that have been completed for at least 30 minutes. Note that once a suspend or shutdown operation is started, the other operations will fail immediately but a task MBean is still created. The `isRunning()` method returns true if suspend or shutdown is still running. Timeout of the operation is controlled by the timeout parameter on the new task operations. If set to 0, the default is used. The default is to use `Inactive Connection Timeout Seconds` if set or 60 seconds. If you want a minimal timeout, set the value to 1. If you want no timeout, set it to a large value (not recommended).

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. For a sample WLST script that shuts down a data source, see the WLST example WLST example.

Starting a Data Source

Use the Start option to start a data source which has been shut down.

The Start option is available in the WebLogic Remote Console or the `start()` method in the `JDBCDataSourceRuntimeMBean`.

```
JDBCDataSourceRuntimeMBean.start
```

Invoking the Start operation re-initializes the data source, creates connections and transitions the data source to a health state of running.

For more information, see *Oracle WebLogic Remote Console Online Help*.

Managing DBMS Network Failures

Manage the DBMS network failures by setting a desired amount of time for `-Dweblogic.resourcepool.max_test_wait_secs=xx` .

Here, `xx` is the amount of time, in seconds, WebLogic Server waits for connection test before considering the connection test failed. By default, a server instance is assigned a value of 10 seconds.

This command line flag manages failures, such as a DBMS network failure, which can cause connection tests and connections in use by applications to hang for extended periods of time (for example, 10 minutes). If the assigned time period expires, the server instance purges unused connections and puts a watch on connections that are in use by the application.

A value of ten seconds provides a reasonable amount of time to allow for peak stress loads, when a DBMS may temporarily halt responses to clients, and then resume service on existing connections. However, if the wait time is too long or too short, add the flag to the `startWebLogic` script used for starting the server with a value that is more appropriate for your environment. Setting the value for the amount of time to zero (0) seconds, causes the server to wait indefinitely on a hanging connection test.

18
Tuning Data Source Connection Pools

Learn how to use connection pool attributes for JDBC data sources to improve application and system performance.

- Increasing Performance with the Statement Cache
  Reusing cached statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks. Cache configurations options include Statement Cache Type and Statement Cache size.
- Initial Capacity Enhancement in the Connection Pool
  Connection retry, early failure, and critical data sources are available from WebLogic Server 12.2.1.3, to enhance the initial capacity connections in the connection pool.
- Connection Testing Options for a Data Source
- Enabling Connection Creation Retries
- Enabling Connection Requests to Wait for a Connection
- Automatically Recovering Leaked Connections
  You can automatically recover leaked connection by specifying values for `Inactive Connection Timeout` in the WebLogic Remote Console.
- Avoiding Server Lockup with the Correct Number of Connections
- 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.
- 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 set the `Pinned To Thread` option on the JDBC data source to `true`.
- Using Unwrapped Data Type Objects
  Disabling wrapping allows applications to use native driver objects directly to provide a significant performance improvement.
- Tuning Maintenance Timers
  Learn about the tunable timer properties `weblogic.jdbc.gravitationShrinkFrequencySeconds` `weblogic.jdbc.harvestingFrequencySeconds` and `weblogic.jdbc.securityCacheTimeoutSeconds` used by WebLogic JDBC.
- JDBC Connection Creation Limits
  Some databases place a limit on the rate of JDBC connections that can be created. For example, 100 connections per second. For large WebLogic Server deployments this rate can be exceeded during server startup, database rolling restarts, failover events, and so on. Even with databases that do not enforce a rate limit, many simultaneous JDBC connection requests can potentially overwhelm the database capacity.

Increasing Performance with the Statement Cache

Reusing cached statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks. Cache configurations options include Statement Cache Type and Statement Cache size.

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.
- **Statement Cache Size**—The number of statements to store in the cache for each connection. The default value is `10`. See Statement Cache Size.

Note:

While using the Oracle JDBC driver, the Oracle JDBC driver prepared statement cache must be used instead of the WLS data source prepared statement cache.

You can use the WebLogic Remote Console to set statement cache options for a data source.

- Statement Cache Algorithms
- Statement Cache Size
- Usage Restrictions for the Statement Cache

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** 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 has a resource cost, such as a cursor, for each open statement. This applies to prepared and callable statements in the statement cache. For example, if you cache too many statements, you may exceed the limit of open cursors on your database server. If you have a data source with 10 connections deployed on 2 servers, and 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.
- Some drivers impose large memory requirements for every open statement. For a server, memory consumption is based on (number of data sources * number of connections * number of statements).
- Some DBMSs may impose limits on the number of statements/cursors per connection.

The statement cache size is dependent on your applications. Ideally it is the total number of every prepared or callable statement made with a connection from the DataSource. One way to approximate the maximum size used by your applications is to set the cache size to a huge number, observe the pool statistics of your application, and then take a value slightly larger than the largest observed value. From a WebLogic DataSource perspective, there is no loss in performance for having a cache size larger than your applications require.

However, having a cache size that is too small negatively impacts performance as the cache turnover can be so high while trying to accommodate new statements that old statements are flushed before they are ever reused. In some cases where you cannot allow a big enough statement cache to hold all or most of your statements, you may find may the reuse rate is so small that your system performs better without a statement cache.

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
- Using setNull In a Prepared Statement
- Statements in the Cache May Reserve Database Cursors
- Other Considerations When Using the Statement Cache

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.

Other Considerations When Using the Statement Cache

When `oracle.jdbc.implicitstatementcachesize` is set in the connection properties of a data source, the WebLogic Server statement cache size is automatically set to zero (0).

There are several cases where special consideration is needed for the statement cache.

- If a data source is configured to use DRCP, the cache is cleared whenever the connection is closed by the application. See Database Resident Connection Pooling.
- When a data source is configured to use JDBC Replay Driver using the JDBC Replay Driver driver, the WebLogic Server statement cache size is automatically set to 0.
- `oracle.jdbc.implicitstatementcachesize` is set in the connection properties of a data source.
- For ease of use and to ensure caching is disabled, WebLogic Server automatically sets the statement cache size value to zero (0).
- When the JDBC 4.0 `setPoolable(false)` method is called for a WebLogic data source that has prepared statement caching enabled, the statement is removed from the cache in addition to calling the method on the driver object.

Initial Capacity Enhancement in the Connection Pool

Connection retry, early failure, and critical data sources are available from WebLogic Server 12.2.1.3, to enhance the initial capacity connections in the connection pool.

Creating the Initial Capacity Connections in the Connection Pool

Whenever a server starts, the data source tries to create the initial capacity connections in the connection pool. Prior to 12.2.1.3, the data source attempted to create initial capacity connections even if some of the connection attempts failed. This can take a long time if one or more of the connection failures take a long time due to unavailability of network or database.

From WebLogic Server 12.2.1.3 onwards, the following changes are available during the creation of the initial capacity connections:

- Connection Retry
- Early Failure
- Critical Data Sources

Connection Retry

There are two connection properties that control retrying the initial connection creation failure:

`weblogic.jdbc.startupRetryCount` — If this property is set and the value is greater than 0, if failure occurs connection creation will be retried based on the value. The default value is 0 (no retry).

`weblogic.jdbc.startupRetryDelaySeconds`— If this property is set and the value is greater than 0 and retry count is set, the connection creation will delay for the specified number of seconds between retries. The default value is 0 (no delay).

Early Failure

The following connection property controls whether or not to continue after connection creation fails:

`weblogic.jdbc.continueMakeResourceAttemptsAfterFailure=true` — If startup retry is enabled, the driver property `weblogic.jdbc.continueMakeResourceAttemptsAfterFailure=true` is ignored and the data source will not continue to create connections after a failure when the server is starting. It will continue create attempts if the data source is deployed or redeployed on a running server.

Critical Data Sources

If a failure occurs while populating the initial capacity connections in the connection pool, the data source is not deployed (it won't be in JNDI so the application will fail to find it) but the server continues to startup and is not marked as unhealthy. In some applications, a data source may be a critical resource such that no useful processing can be done if the data source is not deployed. This can be controlled using a connection property:

`weblogic.jdbc.critical` — If this value is set to true, the managed server fails to boot; this does not apply to the administration server, which is available to process configuration changes. The default value is false, where the server continues to boot without deploying the data source.

Example 18-1 WLST Sample Code

The following WLST sample code fragment illustrates defining a Retry count and delay on a data source.

```
edit()
startEdit()
datasource="dsname"
cd("/JDBCSystemResources/" + datasource + "/JDBCResource/" + datasource + "/
JDBCDriverParams/"
+ datasource + "/Properties/" + datasource)
cmo.createProperty("weblogic.jdbc.startupRetryCount", "5")
cmo.createProperty("weblogic.jdbc.startupRetryDelaySeconds", "10")
save()
activate()
```

Connection Testing Options for a Data Source

Learn about testing the database connections for a data source using the Automatic and Manual testing methods.
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. See Testing Data Sources and Database Connections.

To configure automatic testing options for a data source, you set the following options either through the WebLogic Remote 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 connections, and closes and replaces any faulty connections. You must also set the **Test Table Name**.

- **Test Reserved Connections**—(`TestConnectionsOnReserve` in the `JDBCConnectionPoolParamsBean`) Enable 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. See Database Connection Testing Using Default Test Table Name.
- **Seconds to Trust an Idle Pool Connection**—(`SecondsToTrustAnIdlePoolConnection` in the `JDBCConnectionPoolParamsBean`) Use this option to specify the number of seconds after a connection has been proven to be OK that WebLogic Server trusts 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.
- **Count of Test Failures Till Flush**—(`CountOfTestFailuresTillFlush` in the `JDBCConnectionPoolParamsBean`) Use this option to specify the number of test failures allowed before WebLogic Server closes all connections in the connection pool to minimize the delay caused by further database testing. This parameter minimizes the amount of time allowed for failover when a Multi Data Source member fails. See Minimizing Connection Test Delay After Database Connectivity Loss.
- **Connection Count of Refresh Failures Till Disable**—(`CountOfRefreshFailuresTillDisable` in the `JDBCConnectionPoolParamsBean`) Use this option to specify the number of test failures allowed before WebLogic Server disables the connection pool to minimize the delay in handling the connection request caused by a database failure. See Minimizing Connection Request Delays After Loss of DBMS Connectivity.

See WebLogic Remote Console or see JDBCConnectionPoolParamsBean in the *MBean Reference for Oracle WebLogic Server* for more details about these options.

For instructions to set connection testing options, see *Oracle WebLogic Remote Console Online Help*.

Automatic connection testing options are:

- Database Connection Testing Semantics
- Database Connection Testing Configuration Recommendations
- Database Connection Testing Using Default Test Table Name
- Database Connection Testing Options

Database Connection Testing Semantics

When WebLogic Server tests database connections in a data source, it reserves a connection from the data source, 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 complete a test, the server instance may delay testing on other threads until the abnormally long-running test completes. If that thread hangs too long in connection testing (10 seconds by default), a pool may declare a DBMS connectivity failure, disable itself, and kill all connections, whether unreserved or in application hands. A pool closes all in-test or unused connections, and flags in-use connections to check them later as they may be hanging. After the `Test Frequency Seconds` has passed, WebLogic Server kills any in-use connections that have not progressed.

This is very rare, and is intended to relieve the otherwise interminable hangs that can be caused by network cable disconnects and other problems that can lock any JVM thread which is doing a call in a socket read that the JVM will be unable to break until the OS TCP limit is hit (typically 10 minutes).

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 count(*) 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 sting 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 explained in the following topics:

- Connection Testing When Database Connections are Created
- Periodic Connection Testing
- Testing Reserved Connections
- Minimizing Connection Test Delay After Database Connectivity Loss
- Minimizing Connection Request Delays After Loss of DBMS Connectivity
- Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection

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 the pooled connections that are not currently reserved by applications. 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. The default value is not enabled.

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.

Minimizing 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 terminated. If the data source is configured to test connections on reserve, then when an application requests a database connection, WebLogic Server tests the connection, discovers that the connection is terminated, 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 terminated connection can impose a long delay.

To minimize this delay, WebLogic data sources include logic that considers *all* connections in the data source as terminated 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 terminated 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 `CountOfTestFailuresTillFlush` .

  Note:

  The default value is 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.

  Note:

  This value is overridden by `CountOfTestFailuresTillFlush` value. Actually, the number of test failures before closing all connections follows the count of test failures till flush, that is, `CountOfTestFailuresTillFlush`, which is located in `Connection Pool` parameter of WebLogic Remote Console.

To minimize the delay that occurs during the test of dead database connections, you can set `CountOfTestFailuresTillFlush` attribute on the connection pool. To enable this feature, `TestConnectionsOnReserve` must also be set to `true`.

If the configured or default number of consecutive connection test failures are observed, then all currently unused connections in the pool are terminated so that any subsequent connection requests get a new connection. Active connections are not interrupted but are monitored for activity. If no activity is detected with in 60 seconds, these connections are destroyed.

The default value is generally sufficient. You may need to increase this value if your environment has:

- Slow-running applications that may not show JDBC activity for several minutes

- Network/firewall issues that consistently terminate one or two connections

Minimizing Connection Request Delays After Loss of DBMS Connectivity

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, in cases where the DBMS is truly down, it may be minutes, hours, or days before the DBMS is restored. Testing a dead database connection can take as long as the network timeout, and can cause a long delay for clients. This delay occurs for each dead connection in the connection pool until all connections are replaced and can cause long delays to clients before getting the expected failure message.

To minimize the delay that occurs for client applications while a database is unavailable, you can set the `CountOfRefreshFailuresTillDisable` attribute on the connection pool. The default value is 2. To enable this feature, `TestConnectionsOnReserve` must also be set to `true` and `InitialCapacity` must be greater than 0.

If the configured or default number of consecutive failures to replace a dead connection are observed, WebLogic Server suspends the connection pool. If an application requests a connection from a suspended connection pool, WebLogic Server throws `PoolDisabledSQLException` 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 WebLogic Remote Console or WLST.

> **Note:**
>
> If a data source is added to a Multi Data Source, the Multi Data Source takes over the responsibility of disabling and re-enabling its data sources. By default, a Multi Data Source will check every two minutes (configurable) and re-enable any of its data sources that can re-establish connections. Configure using `test frequency seconds` at the Multi Data Source level. Note that the semantics of this setting are different than at the data source level.

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

For some applications that use DBMS connections in a lot of very short cycles (such as reserve-do_one_query-close), the data source's testing of the connection can contribute a significant amount of overhead to each use cycle. 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 by an application, 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 *Oracle WebLogic Remote 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 to delays in getting a connection from the data source and can tolerate a possible application failure due to using a dead connection, you should set Seconds to Trust an Idle Pool Connection to a higher number, set Test Frequency to a lower number, and enable Test Reserved Connections.

With these settings, your application will rely more on the data source testing connections in the pool when they are not in use, rather than when an application requests a connection.

> **Note:**
>
> Ultimately, even if WebLogic does its best, a connection may fail in the instant after WebLogic successfully tested it, and just before the application uses it. Therefore, every application should be written to respond appropriately in the case of unexpected exceptions from a dead connection.

When running with AGL and FAN enabled:

- It is not necessary to run with `Test Connections on Reserve` because ONS will send *down events* when a database instance goes down. This can significantly improve performance by eliminating (or reducing) testing overhead in the database. However, `Test Connections on Reserve` tests for other failures such as network connectivity and application access to the database. Oracle recommends running with `Test Connections on Reserve` and using `SecondsToTrustAnIdlePoolConnection` and/or `TestFrequencySeconds` to reduce the overhead.
- `CountOfTestFailuresTillFlush` and `CountOfRefreshFailuresTillDisable` are ignored. The disabling an entire RAC instance occurs when a FAN event is received that indicates that the instance is down.

Database Connection Testing Using Default Test Table Name

When you create a data source using the WebLogic Remote Console, the WebLogic Remote 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 Remote Console).

The `Test Table Name` is an overloaded parameter. Its simplest form is to name a table that WebLogic Server queries to test a connection. Setting it to any table, such as "DUAL" for Oracle, causes the data source to run the query `select count(*) from DUAL`. If used in this mode, Oracle recommends that you choose a small, infrequently updated table (preferably a pseudo-table such as DUAL).

The second manner in which you can define this parameter is to allow any specific SQL string to be executed to test the connection. To use this option, set the parameter to "SQL " plus the desired SQL string. For example `SQL select 1` works for SQLServer, which does not need a table in queries to select constants. This option is useful for adding DBMS-side control of WebLogic Server pool connection testing, and to make the test as fast as possible.

Table 18-1 Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
DB2	SQL SELECT COUNT(*) FROM SYSIBM.SYSTABLES
Microsoft SQL Server	SQL SELECT 1
MySQL	SQL SELECT 1
Oracle	SQL ISVALID
Sybase	SQL SELECT 1

Database Connection Testing Options

For applications using an Oracle data base, particularly those with Oracle RAC environments, using the default value of the `Test Table Name` attribute provides the best overall performance.

Oracle continues to support `SQL PINGDATABASE` and `SQL SELECT 1 FROM DUAL`. Although not as thorough as using `SQL SELECT 1 FROM DUAL`, `SQL ISVALID` significantly eliminate processing overhead and improve SOA workload performance.

Enabling Connection Creation Retries

WebLogic JDBC data sources offer the Creation Retry Frequency option, which sets the number of seconds between attempts to establish connections to the database. If you do not set this value, data source creation fails if the database is unavailable. If set and if the database is unavailable when the data source is created, WebLogic Server will attempt 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 JDBCConnectionPoolParamsBean in the *MBean Reference for Oracle WebLogic Server* .

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.

- Connection Reserve Timeout
- Limiting the Number of Waiting Connection Requests

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.

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.

See JDBCConnectionPoolParamsBean in the *MBean Reference for Oracle WebLogic Server*.

Automatically Recovering Leaked Connections

You can automatically recover leaked connection by specifying values for `Inactive Connection Timeout` in the WebLogic Remote Console.

A leaked connection is a connection that was not properly returned to the connection pool in the data source. When you set a value for `Inactive Connection Timeout`, WebLogic Server forcibly returns 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.

For more details about this option, see JDBCConnectionPoolParamsBean in the *MBean Reference for Oracle WebLogic Server*.

Note:

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

To avoid receiving an error while attempting to get a connection from a data source in which there are no available connections, 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.

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. WebLogic Server will make the call, and if the driver throws an exception, the value will be ignored. In some cases, the driver may silently not support the call, or may document limited support. Oracle recommends that you check the driver documentation to verify the expected behavior.

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

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 set the `Pinned To Thread` option on the JDBC data source 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:**
>
> The `Pinned To Thread` feature does not work with an `IdentityPool`. Starting with WebLogic Server Release 12.1.2, configurations with this combination will cause the datasource to fail to deploy.

- Changes to Connection Pool Administration Operations When PinnedToThread is Enabled
- Additional Database Resource Costs When PinnedToThread is Enabled

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.

Consider the following when using the `PinnedToThread` feature:

- If used with `Identity Based Connection Pooling Enabled` set to `true`, an error is thrown and the data source will not deploy.
- When used with `Use Database Credentials` set to `true`, all connections are owned by the default user as defined in the JDBC descriptor but the Oracle proxy is set to the user and password specified on `getConnection(user, password)`. Similarly, with `Oracle Proxy` set to `true`, the user and password are mapped to a database credential and the Oracle proxy is set. This is the same behavior as without `PinnedToThread`.
- Connection labeling is not supported when using `PinnedToThread` and an exception is thrown when trying to get a connection with label properties.
- When using Multi Data Source (MDS), connections are maintained by each member data source as they are selected by the MDS. For example, with Algorithm Type of `Failover`, connections are initially be maintained only for the primary member of MDS. If a failover occurs, then connections are maintained for the next member of the MDS. When used with

the Algorithm Type of `Load-Balancing`, connections are maintained for each member of the MDS.

- When using Active GridLink, `Affinity` and `Runtime Load Balancing` continue to work as before with regard to choosing an instance. As many as one connection is stored per instance per thread (the equivalent of setting `OnePinnedConnectionOnly=true` but on a per instance basis). Gravitation is not supported (no migration of connections to lightly used nodes).

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="onePinnedConnectionOnly=true;user=examples"
```

If your system can handle the additional resource requirements, Oracle 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`, Oracle recommends *not* using `PinnedToThread`.

Using Unwrapped Data Type Objects

Disabling wrapping allows applications to use native driver objects directly to provide a significant performance improvement.

Some JDBC objects from a driver that are returned from WebLogic Server are wrapped by default. Wrapping data source objects provides WebLogic Server the ability to:

- Generate debugging output from all method calls.
- Track connection utilization so that connections can be timed out appropriately.
- Provide transparent automatic transaction enlistment and security authorization.

WebLogic Server provides the ability to disable the wrapping of some objects which provides the following benefits:

- Although WebLogic Server generates a dynamic proxy for vendor methods that implement an interface to show through the wrapper, some data types do not implement an interface. For example, Oracle data types Array, Blob, Clob, NClob, Ref, SQLXML, and Struct are classes that do not implement interfaces. Disabling wrapping allows applications to use native driver objects directly.

> **Note:**
>
> Oracle recommends not using these concrete classes and instead using standard SQL types or corresponding Oracle interfaces. See Using API Extensions for Oracle JDBC Types in *Developing JDBC Applications for Oracle WebLogic Server*.

- Eliminating wrapping overhead can provide a significant performance improvement.

When wrapping is disabled (the `wrap-types` element is `false`), the following data types are not wrapped:

- Array
- Blob
- Clob
- NClob
- Ref
- SQLXML
- Struct
- ParameterMetaData
  - No connection testing performed
- ResultSetMetaData
  - No connection testing performed
  - No result set testing performed
  - No JDBC MT profiling performed
- How to Disable Wrapping

How to Disable Wrapping

You can use the WebLogic Remote Console and WLST to disable data type wrapping.

- Disable Wrapping using the Remote Console
- Disable Wrapping using WLST

Disable Wrapping using the Remote Console

To disable wrapping of JDBC data type objects:

1. In WebLogic Remote Console, click **Edit Tree**.
2. In the left pane of the console, expand **Services**, 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. Click **Advanced** tab to show the advanced connection pool options.
6. In **Wrap Data Types**, deselect the checkbox to disable wrapping.
7. Click **Save**.

8. To activate these changes, select **Shopping Cart** and click **Commit Changes**.

   This change does not take effect immediately—it requires that the data source be redeployed or the server be restarted.

Disable Wrapping using WLST

The following is a WLST code snippet to disable data type wrapping:

```
. . .
jdbcSR = create(dsname,"JDBCSystemResource");
theJDBCResource = jdbcSR.getJDBCResource();
poolParams = theJDBCResource.getJDBCConnectionPoolParams();
poolParams.setWrapTypes(false);
. . .
```

This change does not take effect immediately—it requires that the data source be redeployed or the server be restarted.

Tuning Maintenance Timers

Learn about the tunable timer properties `weblogic.jdbc.gravitationShrinkFrequencySeconds` `weblogic.jdbc.harvestingFrequencySeconds` and `weblogic.jdbc.securityCacheTimeoutSeconds` used by WebLogic JDBC.

- `weblogic.jdbc.gravitationShrinkFrequencySeconds`—Connections may be shut down periodically on Active GridLink data sources. If the connections allocated to various RAC instances do not correspond to the Runtime Load Balancing percentages in the FAN load-balancing advisories, connections to overweight instances are destroyed and new connections opened. This process occurs every 30 seconds by default. You can tune this behavior using the `weblogic.jdbc.gravitationShrinkFrequencySeconds` system property which specifies the amount of time, in seconds, the system waits before rebalancing connections. A value less than or equal to 0 disables the rebalancing process.
- `weblogic.jdbc.harvestingFrequencySeconds`—Connection harvesting releases reserved connections that are marked harvestable by the application when a data source falls to a specified number of available connections. This check by default is done every 30 seconds. This system property can be used to change the frequency of harvesting by Data Source the amount of time, in seconds. If set less than or equal to 0, connection harvesting is turned off. See Recover Harvested Connections.
- `weblogic.jdbc.securityCacheTimeoutSeconds`—Performance is impacted when reserving connections from a connection pool, due to the credentials for the WebLogic server user being checked for each reserve connection request. To resolve this, checking can be controlled by this system property. If less than or equal to zero, the cache is turned off and user authentication happens each time. If greater than zero, user authentication is done only once for each user in the specified time period in seconds; the value is then cached. In situations where pool access restrictions are dynamically altered, the pool re-authenticates the users once each time after the cache is cleared. The default value is 10 minutes.

JDBC Connection Creation Limits

Some databases place a limit on the rate of JDBC connections that can be created. For example, 100 connections per second. For large WebLogic Server deployments this rate can be exceeded during server startup, database rolling restarts, failover events, and so on. Even

with databases that do not enforce a rate limit, many simultaneous JDBC connection requests can potentially overwhelm the database capacity.

To avoid JDBC driver connection errors, the data source connection property `weblogic.jdbc.maxConcurrentCreateRequests` can be used to limit the number of concurrent connection-create operations. The `weblogic.jdbc.concurrentCreateRequestsTimeoutSeconds` property can be used to specify how long a connection create request waits (60 seconds by default) for a permit to proceed.

A

Configuring JDBC Application Modules for Deployment

Learn how to package and scope a data source for use in enterprise applications and the details of packaged JDBC modules.

Note:

To learn more about the proprietary mechanism provided by WebLogic Server prior to the DatasourceDefinition feature introduced in Jakarta EE 8, see Using Java EE DataSources Resource Definitions in *Developing JDBC Applications for Oracle WebLogic Server*.

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*).

- Packaging a JDBC Module with an Enterprise Application: Main Steps
  Learn about the steps for creating, packaging, and deploying a JDBC module with an enterprise application.
- Creating Packaged JDBC Modules
- Referencing a JDBC Module in Jakarta EE Descriptor Files
- Packaging an Enterprise Application with a JDBC Module
- Deploying an Enterprise Application with a JDBC Module
- Getting a Database Connection from a Packaged JDBC Module

Packaging a JDBC Module with an Enterprise Application: Main Steps

Learn about the steps for creating, packaging, and deploying a JDBC module with an enterprise application.

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.
2. Add references to the module in all applicable descriptor files. See Referencing a JDBC Module in Jakarta EE Descriptor Files.

3. Package all application modules in an EAR. See Packaging an Enterprise Application with a JDBC Module.
4. Deploy the application. See Deploying an Enterprise Application with a JDBC Module.

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 WebLogic Remote Console.

> **Note:**
>
> You can create a JDBC data source using the WebLogic Remote 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. Modules that represent a generic or Active GridLink data source include all of the configuration parameters for the Generic or Active GridLink data source. Modules that represent a Multi Data Source include configuration parameters for the Multi Data Source, including a list of Generic data source modules used by the Multi Data Source.

- Creating a JDBC Data Source Module Using the Remote Console
- JDBC Packaged Module Requirements
- JDBC Application Module Limitations
- Creating a Generic Data Source Module
- Creating an Active GridLink Data Source Module
- Creating a Multi Data Source Module
- Encrypting Database Passwords in a JDBC Module
- Application Scoping for a Packaged JDBC Module

Creating a JDBC Data Source Module Using the Remote Console

To create a data source module in the WebLogic Remote 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. 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.

4. Continue with adding references to the descriptor files in the enterprise application. See Referencing a JDBC Module in Jakarta EE Descriptor Files.

JDBC Packaged Module Requirements

A JDBC module must meet the following criteria:

- Conforms to the `jdbc-data-source.xsd` schema. The schema is available at http://www.oracle.com/webfolder/technetwork/weblogic/jdbc-data-source/index.html.
- 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 `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 Generic 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 `jdbc-data-source.xsd` schema for more valid entries. For an explanation of each element, see JDBCDriverParamsBean in the *MBean Reference for Oracle WebLogic Server*.
- `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-javaee.xsd` schema, including `initial-capacity`, `min-capacity`, `max-capacity`, and other options common to pooled resources. For more information, see the following:
  - JDBCConnectionPoolParamsBean in the *MBean Reference for Oracle WebLogic Server*
  - `jdbc-data-source.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 `jdbc-data-source.xsd` schema for more valid entries. For an explanation of each element, see JDBCDataSourceParamsBean in the *MBean Reference for Oracle WebLogic Server*.
- `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 *MBean Reference for Oracle WebLogic Server*.

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

Example A-1 Sample Generic Data Source Module

```
<jdbc-data-source xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/domain.xsd"
 xmlns="http://xmlns.oracle.com/weblogic/jdbc-data-source"
 xmlns:sec="http://www.bea.com/ns/weblogic/90/security"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:wls="http://www.bea.com/ns/weblogic/90/security/wls">
  <name>examples-demoXA-2</name>
  <jdbc-driver-params>
    <url>jdbc:derby://localhost:1527/examples;create=true</url>
    <driver-name>org.apache.derby.jdbc.ClientXADataSource</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>examples</value>
      </property>
      <property>
        <name>DatabaseName</name>
        <value>examples</value>
      </property>
    </properties>
    <password-encrypted>{AES}MEK6bPum8M69KRP4FANx3TG/O0iSWRYu2rZGUwnVo6U=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <max-capacity>100</max-capacity>
    <connection-reserve-timeout-seconds>25</connection-reserve-timeout-seconds>
    <test-table-name>SQL SELECT 1 FROM SYS.SYSTABLES</test-table-name>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <global-transactions-protocol>TwoPhaseCommit</global-transactions-protocol>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Creating an Active GridLink Data Source Module

Active GridLink data source modules are similar to Generic data source system modules. Active GridLink data sources include an `jdbc-oracle-params` section that includes `ONS` and `FAN`.

Creating a Multi Data Source Module

A JDBC Multi Data Source module is much simpler than a Generic 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 *MBean Reference for Oracle WebLogic Server*.

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

Example A-2 Sample JDBC Multi Data Source Module

```
<jdbc-data-source xmlns="http://xmlns.oracle.com/weblogic/jdbc-data-source">
  <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

Oracle 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 *WLST Command Reference for Oracle WebLogic Server*.

- Deploying JDBC Modules to New Domains

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.
- Re-encrypt the passwords for your new domain. See Encrypting Database Passwords in a JDBC Module.
- If you use the Oracle wallet, you can simply reference the wallet and copy the wallet file to the new domain. See Creating and Managing Oracle Wallet.

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

Referencing a JDBC Module in Jakarta EE Descriptor Files

Learn about referencing a JDBC Module in Jakarta 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

Enterprise Application (myapp.ear)

application.xml

```
<module>
  <ejb>
  ejbs/Bean1.jar
  </ejb>
</module>
```

weblogic-application.xml

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

META-INF

ejb-jar.xml

```
<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>
```

weblogic-ejb-jar.xml

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

EJB

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

data-source-1-jdbc.xml

```
<jdbc-data-source>
  <name>data-source-1</name>
  <jdbc-driver-params>
  ...
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
  ...
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>qa-database-1
    </jndi-name>
    <scope>Application</scope>
  </jdbc-data-source-params>
  <jdbc-xa-params>
  ...
  </jdbc-xa-params>
</jdbc-data-source>
```

- Packaged JDBC Module References in weblogic-application.xml
- Packaged JDBC Module References in Other Descriptors

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 Jakarta 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

Learn about packaging an application with a JDBC module.
See Packaging Applications Using wlpackage in *Developing Applications for Oracle WebLogic Server*.

Deploying an Enterprise Application with a JDBC Module

Learn about deploying an enterprise application with a JDBC module.
See Deploying Applications Using wldeploy in *Developing Applications for Oracle WebLogic Server*.

Note:

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 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();
```

B

Using Multi Data Sources with Oracle RAC

Learn how to configure and use Multi Data Sources on Oracle Real Application Clusters (RAC) with WebLogic Server. Oracle continues to support Multi Data Sources configurations for legacy application environments using 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 section 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.

Note:

> Oracle recommends using Active GridLink data sources when developing new Oracle RAC applications and when legacy applications do not use Multi Data Sources. See Using WebLogic Server with Oracle RAC.

- Overview of Oracle RAC
  Oracle 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. Oracle RAC comprises two or more Oracle database instances running on two or more clustered machines and accessing a shared storage device via cluster technology.
- Software Requirements
  Learn about the software required to use WebLogic Server with Oracle RAC.
- JDBC Driver Requirements
  To use WebLogic Server with Oracle RAC, your WebLogic generic data sources must use the Oracle JDBC Thin driver 11g or later to create database connections.
- Hardware Requirements
  A typical WebLogic Server/Oracle RAC system includes a WebLogic Server cluster, an Oracle RAC cluster, and hardware for shared storage.
- Configuring Multi Data Sources with Oracle RAC
  When using Multi Data Sources with Oracle RAC, you must configure your WebLogic Domain so that it can interact with Oracle RAC instances and so that it performs as expected.
- 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. Individual transactions are load balanced with the initial connection request for the transaction.
- Using Multi Data Sources without Global Transactions
  Learn about the configurations that use Oracle RAC with Multi Data Sources in an application that does not require global transactions.

- Configuring Connections to Services on Oracle RAC Nodes
  If you rely on Oracle services in your Oracle RAC cluster for workload management, you must use Multi Data Sources to connect to those services instead of you using a Service ID (SID). A WebLogic Server Generic data source can be configured to connect only to a specific service on a specific Oracle RAC node, providing both workload management and load balancing.
- Using SCAN Addresses with Multi Data Sources
  Use Single Client Access Name (SCAN) for providing connection to time listener failover and load-balancing.
- XA Considerations and Limitations when using Multi Data Sources with Oracle RAC
  Learn about the certain requirements and limitations you need to consider when using XA (Global transactions) withMulti Data Sources on Oracle RAC.
- 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.

Overview of Oracle RAC

Oracle 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. Oracle 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, Oracle Automatic Storage Management (ASM), or compatible third party clustering software. Figure B-1 shows an Oracle RAC configuration.

Figure B-1 Oracle Real Application Clusters Configuration

RAC Node 1
RAC Node 2
Interconnect
Shared Storage

Oracle RAC offers features in the following areas:

- Oracle RAC Scalability with WebLogic Server Multi Data Sources

- Oracle RAC Availability with WebLogic Server Multi Data Sources
- Oracle RAC Load Balancing with WebLogic Server Multi Data Sources

Oracle RAC Scalability with WebLogic Server Multi Data Sources

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 Oracle 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 Oracle RAC nodes, storage, or both.

As the number of nodes in an Oracle RAC increases, you scale the number of generic data sources by the number of nodes added to the Oracle RAC. This requires a complex configuration (requiring n+1 data sources where n is the number of Generic data sources plus a Multi Data Source) that requires administrative intervention when the Oracle RAC topology changes.

Oracle RAC Availability with WebLogic Server Multi Data Sources

A Multi Data Source provides an ordered list of generic data sources to use to satisfy connection requests. Normally, every connection request to this kind of Multi Data Source is served by the first Generic data source in the list. If a database connection test fails and the connection cannot be replaced, or if the Generic data sources is suspended, a connection is sought sequentially from the next Generic data source on the list. See Failover and Multi Data Source-Managed Failover and Load Balancing.

Oracle RAC Load Balancing with WebLogic Server Multi Data Sources

Multi Data Sources provide load balancing for XA and non-XA environments. TheGeneric data sources that form a multi data sourceMulti Data Source are accessed using a round-robin scheme. When switching connections, WebLogic Server selects a connection from the next Generic data source in the order listed.

Software Requirements

Learn about the software required to use WebLogic Server with Oracle RAC.

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

- Operating system patches required to support Oracle RAC. See the release notes from Oracle for details.
- Oracle 11g 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 Oracle Automated Storage Management (ASM). Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

> **Note:**
>
> See Supported Configurations in *What's New in Oracle WebLogic Server* 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.

JDBC Driver Requirements

To use WebLogic Server with Oracle RAC, your WebLogic generic data sources must use the Oracle JDBC Thin driver 11g or later to create database connections.

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
- Oracle RAC Cluster
- Shared Storage

WebLogic Server Cluster

The WebLogic Server cluster can be configured in many ways and with various hardware options. See *Administering Clusters for Oracle WebLogic Server* 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 Oracle 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.

Configuring Multi Data Sources with Oracle RAC

When using Multi Data Sources with Oracle RAC, you must configure your WebLogic Domain so that it can interact with Oracle RAC instances and so that it performs as expected.

- Choosing a Multi Data Source Configuration for Use with Oracle RAC
- Configuring Multi Data Sources for use with Oracle RAC
- Configuration Considerations for Failover
- Configuring the Listener Process for Each Oracle RAC Instance
- Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled
- Additional Configuration Considerations

Choosing a Multi Data Source Configuration for Use with Oracle RAC

WebLogic Server Multi Data Sources support several configuration options for using Oracle RAC:

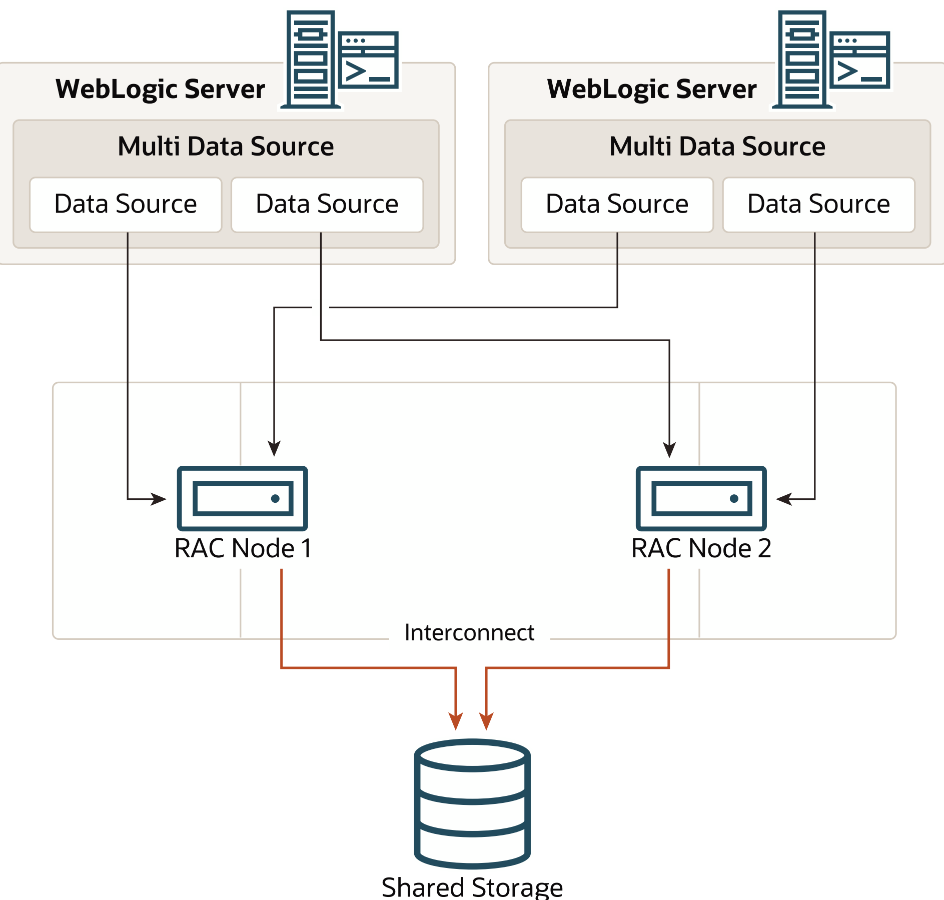
- To connect to multiple Oracle RAC instances when using Global transactions (XA), see Using Multi Data Sources with Global Transactions.
- To connect to multiple Oracle RAC instances when not using XA, see Using Multi Data Sources without Global Transactions.
- You can also configure Multi Data Sources to connect to specific services that are running on Oracle RAC nodes. Both XA and non-XA drivers are supported, see Configuring Connections to Services on Oracle RAC Nodes.

Configuring Multi Data Sources for use with Oracle RAC

To connect WebLogic Server to multiple Oracle RAC nodes using Multi Data Sources, first configure a Generic data source for each Oracle RAC instance in your Oracle 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 generic data sources to it.

Figure B-2 shows a typical Multi Data Source configuration.

Figure B-2 Multi Data Source Configuration



You can use the WebLogic Remote Console or any other means that you prefer to configure your domain, such as the WebLogic Scripting Tool (WLST) or a JMX program. For information about configuring a WebLogic JDBC Multi Data Source see Configuring JDBC Multi Data Sources. For information on how to configure the Generic data sources in a Multi Data Source to connect to services running on Oracle RAC nodes, see Configuring Connections to Services on Oracle RAC Nodes.

To use a database connection in this configuration, your applications look up one v on the JNDI tree and then request a connection. The Multi Data Source determines which generic 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

Attributes of a Multi Data Source

The multi data source may have the following attributes, depending on the role of Oracle 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 generic data sources; with the `Failover` option, connection requests are served by the first available pool in the list. When a generic data source becomes defunct, connection requests are served by the next generic data source in the list.

- `FailoverRequestIfBusy="true"`

  With the Failover algorithm, this attribute enables failover when all connections in a generic data source are in use.

- `TestFrequencySeconds="120"`

  This attribute controls the frequency at which WebLogic Server checks the health of generic data sources previously marked as unhealthy to see if connections can be recreated and if the generic data source can be re-enabled. For more details see Configuring JDBC Multi Data Sources.

  For fast failover of Oracle RAC nodes, set this value to a smaller interval, for example, `10` (seconds).

Configuration Considerations for Failover

Consider the following information when configuring for failover.

- Multi Data Source-Managed Failover and Load Balancing
- Delays During Failover
- Failure Handling Walkthrough for Global Transactions

Multi Data Source-Managed Failover and Load Balancing

Multi Data Sources offer failover and load balancing for global transactions. For a description of Multi Data Source failover features, see Multi Data Source Failover Enhancements.

With this configuration, pictured in Figure B-2, you get:

- Fast 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 an Oracle 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 Generic data source and routes connection requests to other Generic data sources (which correspond to other Oracle RAC nodes) in the Multi Data Source. WebLogic Server periodically tries to recreate the database connections in the disabled Generic data source. When WebLogic Server is successful in recreating the connections, it next re-enables the Generic data source and begins routing connection requests to the Generic 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.

Delays During Failover

Occasionally, when one Oracle 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`.

When a server acting as Coordinator returns to service, it takes the following actions during recovery:

- The Transaction Manager reads the transaction checkpoints and the resource checkpoints from the TLog.

- The transactions read from the TLOG (transaction checkpoints) become active and the state is set to committing. The TM tries to commit these transactions just as it does for other runtime transactions. If the commit fails a retry-commit process takes place until `AbandonTimeoutSeconds` after a grace period has expired.
- The TM calls `xa.recover` on resources read from the TLOG (resource checkpoints) to obtain a list of pending transactions. If the `xa.recover` call fails, the TM retries the `xa.recover` call on the resource every `XARetryIntervalSeconds` for a period of `XARetryDurationSeconds`.

Use the following formula to determine the value for `XARetryDurationSeconds`:

`XARetryDurationSeconds` = (longest transaction timeout for transactions that use connections from the generic data source) + (delay before XIDs are available on all Oracle 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 WebLogic Remote Console, use the following steps:

1. In WebLogic Remote Console, click **Edit Tree**.
2. In the left pane of the console, expand **Services > JDBC**, then select **Data Sources**.
3. On the Summary of Data Sources page, click the data source name.
4. Select the **Transaction** tab.
5. Update `XA Retry Duration` and `XA Retry Interval`.
6. Click **Save**.

Optionally, you can use WebLogic Scripting Tool (WLST) or a JMX program.

Failure Handling Walkthrough for Global Transactions

What happens to in-flight 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 Oracle 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.

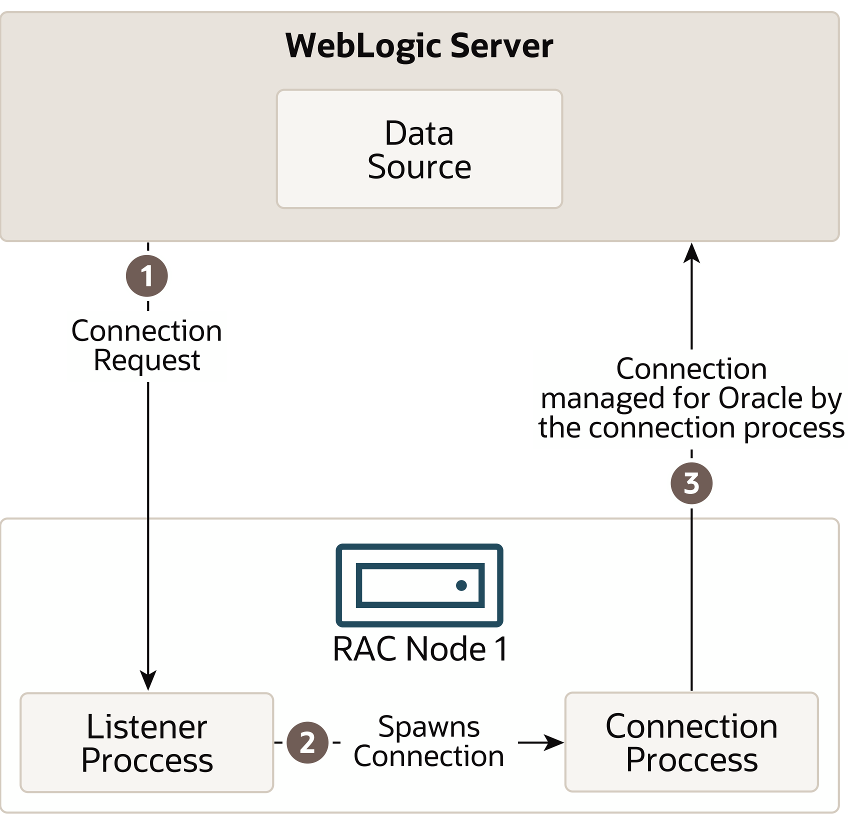
Configuring the Listener Process for Each Oracle RAC Instance

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 data source.

When a multi 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 generic 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-3 shows the Oracle listener process functionality.

Figure B-3 Oracle Listener Process Functionality



To enable this functionality, you have two options:

- **Use local listeners.** Configure the listener process for each Oracle RAC instance in the Oracle cluster. WebLogic Server requires that you configure a local listener on each Oracle RAC instance. Each database instance should be configured to register *only* with its local listener.

  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. Oracle recommends using dynamic registration.

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

- **Use remote listeners.** WLS requires that you specify both the SERVICE_NAME and the INSTANCE_NAME in the JDBC URL for the generic data sources in the multi data source. See Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled.

Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled

If the server-side load balancing feature has been enabled for the Oracle RAC backend (using remote_listeners), the JDBC URL that you use in the generic data sources of a multi data source configuration should include the INSTANCE_NAME. For example, the URL can be specified in the following format:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=host-vip)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=dbservice)(INSTANCE_NAME=inst1)))
```

If specifying the INSTANCE_NAME in the URL is not possible, remote listeners must be disabled. To disable remote listeners, delete any listed remote listeners in `spfile.ora` on each Oracle RAC node. For example:

```
*.remote_listener="
```

In this case, the recommended URL that you use in the generic data sources of a multi data source configuration is:

```
jdbc:oracle:thin:@host-vip:port/dbservice
```

or

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=host-vip)(PORT=1521))(
CONNECT_DATA=(SERVICE_NAME=dbservice)))
```

Additional Configuration Considerations

In some deployments of Oracle RAC, you may need to set parameters in addition to the out of the box configuration of a data source in an Oracle RAC configuration. The additional parameters are:

- Set `oracle.jdbc.ReadTimeout=300000` (300000 milliseconds) for each generic data source.

  The actual value of the `ReadTimeout` parameter used may differ based on your application environment.

- If the network is not reliable, it is difficult for a client to detect the frequent disconnections when the server is abruptly disconnected. By default, a client running on Linux takes 7200 seconds (2 hours) to sense the abrupt disconnections. This value is equal to the value of the `tcp_keepalive_time` property. To configure the application to detect the disconnections faster, set the value of the `tcp_keepalive_time`, `tcp_keepalive_interval`, and `tcp_keepalive_probes` properties to a lower value at the operating system level.

  Note:

  Setting a low value for the `tcp_keepalive_interval` property leads to frequent probe packets on the network, which can make the system slower. Set the value of this property based on system requirements of your application environment.

  For example, set `tcp_keepalive_time=600` for a system running a WebLogic Server managed server.

- Specify the `ENABLE=BROKEN` parameter in the `DESCRIPTION` clause in the connection descriptor. For example:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(enable=broken)
  (ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=node1-vip.mycompany.com)
  (PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=orcl.country.myCorp.com)
  (INSTANCE_NAME=orcl1)))
  ```

The following code snippet provides an example generic data source configuration:

```
<url>jdbc:oracle:thin:@(DESCRIPTION=(enable=broken)(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
(HOST=node1-vip.country.myCorp.com)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME=orcl.country.myCorp.com)(INSTANCE_NAME=orcl1)))</url>
<driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
```

```
<properties>
<property>
<name>oracle.jdbc.ReadTimeout</name>
<value>300000</value>
</property>
<property>
<name>user</name>
<value>jmsuser</value>
</property>
<property>
<name>oracle.net.CONNECT_TIMEOUT</name>
<value>10000</value>
</property>
</properties>
```

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. Individual transactions are load balanced with the initial connection request for the transaction.

Failover is handled at the Multi Data Source level when a Oracle RAC instance becomes unavailable. If there is a failure on a Oracle RAC instance before PREPARE, the transaction is lost. 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
- Required Attributes of Data Sources within a Multi Data Source Using Global Transactions
- Sample Configuration Code

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 generic data source. The attributes include the following:
  - `XARetryDurationSeconds`
  - `SupportsLocalTransaction`
  - `KeepXAConnTillTxComplete`
  - `NeedTxCtxOnClose`
  - `XAEndOnlyOnce`
  - `NewXAConnForCommit`
  - `RollbackLocalTxUponConnClose`
  - `RecoverOnlyOnce`
  - `KeepLogicalConnOpenOnRelease`

> **Note:**
>
> If you are not using Active GridLink data sources, Oracle recommends the use of Multi Data Sources for failover and load balancing across Oracle RAC instances for XA and non-XA environments. For more information on using Multi Data Sources in non-XA environments, see Using Multi Data Sources without Global Transactions.

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

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

- Oracle JDBC Thin driver. For example:

  ```
  <url>jdbc:oracle:thin:@host1:1521:SNRAC1</url>
  <driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
  ```

- `KeepXAConnTillTxComplete="true"`
  - Forces the generic 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 generic data source. See Test Connections on Reserve to Enable Fail-Over for more details about this attribute.
  - Required to enable failover to another Oracle 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.

Sample Configuration Code

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

```
<jdbc-data-source xmlns="http://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/domain.xsd">
  <name>oracleRACXAPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@host1:1521: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://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/domain.xsd">
  <name>oracleRACXAPool2</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@host2:1521: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://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/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

Learn about the configurations that use 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
- Sample Configuration Code

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

Generic data sources must have the following attributes:

- Oracle JDBC Thin driver. For example:

  ```
  <url>jdbc:oracle:thin:@host1:1521:SNRAC1</url>
  <driver-name>oracle.jdbc.OracleDriver</driver-name>
  ```

- `TestConnectionsOnReserve="true"`
  - Enables testing of a database connection when an application reserves a connection from the Generic data source. Test Connections on Reserve to Enable Fail-Over for more details about this attribute.
  - Required to enable failover and connection request routing within a Multi Data Source (effectively, failover to another Oracle 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.

Sample Configuration Code

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

```
<jdbc-data-source xmlns="http://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/domain.xsd">
  <name>jdbcPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@host1:1521: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://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/domain.xsd">
  <name>jdbcPool2</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@host2:1521: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://xmlns.oracle.com/weblogic/jdbc-data-source"
  xmlns:sec="http://xmlns.oracle.com/weblogic/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wls="http://xmlns.oracle.com/weblogic"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/domain/1.0/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.

Configuring Connections to Services on Oracle RAC Nodes

If you rely on Oracle services in your Oracle RAC cluster for workload management, you must use Multi Data Sources to connect to those services instead of you using a Service ID (SID). A WebLogic Server Generic data source can be configured to connect only to a specific service on a specific Oracle RAC node, providing both workload management and load balancing.

In general, to connect to Oracle RAC services, you need to:

- Create a Multi Data Source for each service to which you want to connect.
- Within each Multi Data Source, create one Generic data source for each Oracle RAC node in the cluster on which the service will be configured, whether or not the service will be actively running on each node.

Configuring a Data Source to Connect to a Service, describes special considerations for configuring these data sources. Service Connection Configurations, shows example configurations for either load balancing or workload management.

- Configuring a Data Source to Connect to a Service
- Service Connection Configurations
- Connection Pool Capacity Planning

Configuring a Data Source to Connect to a Service

You configure a Generic data source to connect to a service running on an Oracle RAC node in the same way as you configure any Generic data source (using WLST, the WebLogic Remote Console, or the Configuration Wizard), with the following exceptions:

- `initial-capacity="0"`

  This prevents pool creation failure for inactive pools at WLS startup, and enables WLS to create the Generic data source even if it can't connect to the service on the node. Without setting this option to `0`, Generic data source creation will fail and the server may fail to boot normally.

  In the WebLogic Remote Console, edit the generic data source after creating it, and set **Initial Capacity** to 0.

- Oracle JDBC Thin (or Thin XA) driver. For example:

  For non-XA:

  ```
  driver-name="oracle.jdbc.OracleDriver"
  url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=RAC1)
  (PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=Service_1)(INSTANCE_NAME=DB_02)))"
  ```

  If configuring via the WebLogic Remote Console, select **Oracles's Driver (Thin) for RAC Service-Instance connections** from the **Database Driver** drop-down and specify the service in the **Service Name** field.

  For XA:

  ```
  driver-name="oracle.jdbc.xa.client.OracleXADataSource"
  url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=RAC1)
  (PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=Service1)(INSTANCE_NAME=DBase1)))"
  ```

If configuring via the WebLogic Remote Console, select **Oracle's Driver (Thin XA) for RAC Service-Instance connections** from the **Database Driver** drop-down and specify the service in the **Service Name** field.

> **Note:**
>
> The `SERVICE_NAME` must be the same for all Generic data sources in a given Multi Data Source.
>
> Specify a different `HOST NAME` and/or port for each Generic data source in a given Multi Data Source.

- When specifying `max-capacity` (**Maximum Capacity** in the WebLogic Remote Console) for the connection pool, you need to consider the connection capacity of each of the Oracle RAC nodes in your configuration, and the total number of connections from all Generic data sources. See Connection Pool Capacity Planning, for more information.

Selecting the Appropriate Multi Data Source Algorithm

For service connection scenarios, Oracle recommends that you configure your Multi Data Source with the **Load Balancing** algorithm. If the Multi Data Source is configured with the **Load Balancing** algorithm, its connection pools are used in a round robin fashion. In this case, workload is load-balanced across all of the Oracle RAC nodes on which the associated service is currently active.

If the Multi Data Source is configured with the **Failover** algorithm, the first Multi Data Source is used to connect to the service on its associated Oracle RAC node, until a connection attempt fails for any reason (for example, the Oracle RAC node becomes unavailable or there are no more connections available in the Multi Data Source). At that point, the second Multi Data Source is used to connect to the service on its associated Oracle RAC node, and so on. In this case, the Oracle RAC node to which the first Multi Data Source is connected will experience more use than the remaining nodes on which the service is running.

Service Connection Configurations

You can design your configuration to provide:

- Workload Management
- Load Balancing

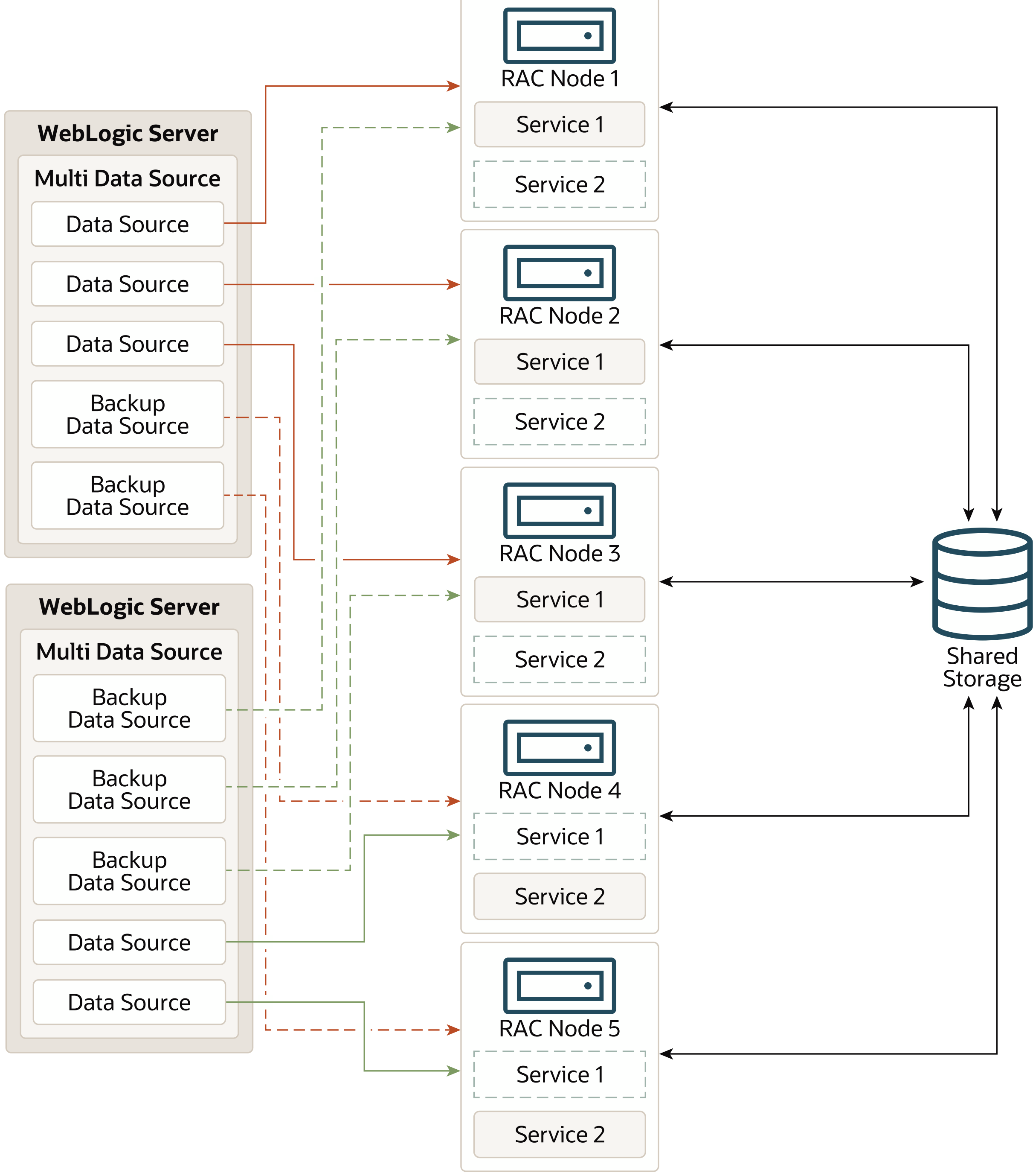
Workload Management

In a workload management configuration, each Multi Data Source has one Generic data source configured for a given service on each Oracle RAC node, regardless of whether the service you are connecting to is active or inactive on a given Oracle RAC node. This lets you quickly start an inactive service on a node and create connections to that service should another node become unavailable due to unplanned downtime or scheduled maintenance. It also lets you quickly increase or decrease the available capacity for a given service based on workload demands.

When you start the service on a node, the associated Generic data source detects that the service is now active, and the Generic data source will then start making connections to that node as needed. When you stop a service on a given node, the associated Generic data source can no longer make connections to that node, and will become inactive until the service is restarted on that node.

The WLS Generic data source performs connection testing. This lets the Generic data source adjust to changes in the topology of the Oracle RAC configuration. The Generic data source performs polling to see if its associated service is active or inactive. The connection test fails if the service is no longer available on the Oracle RAC node.

Figure B-4 Workload Management using Multi Data Sources



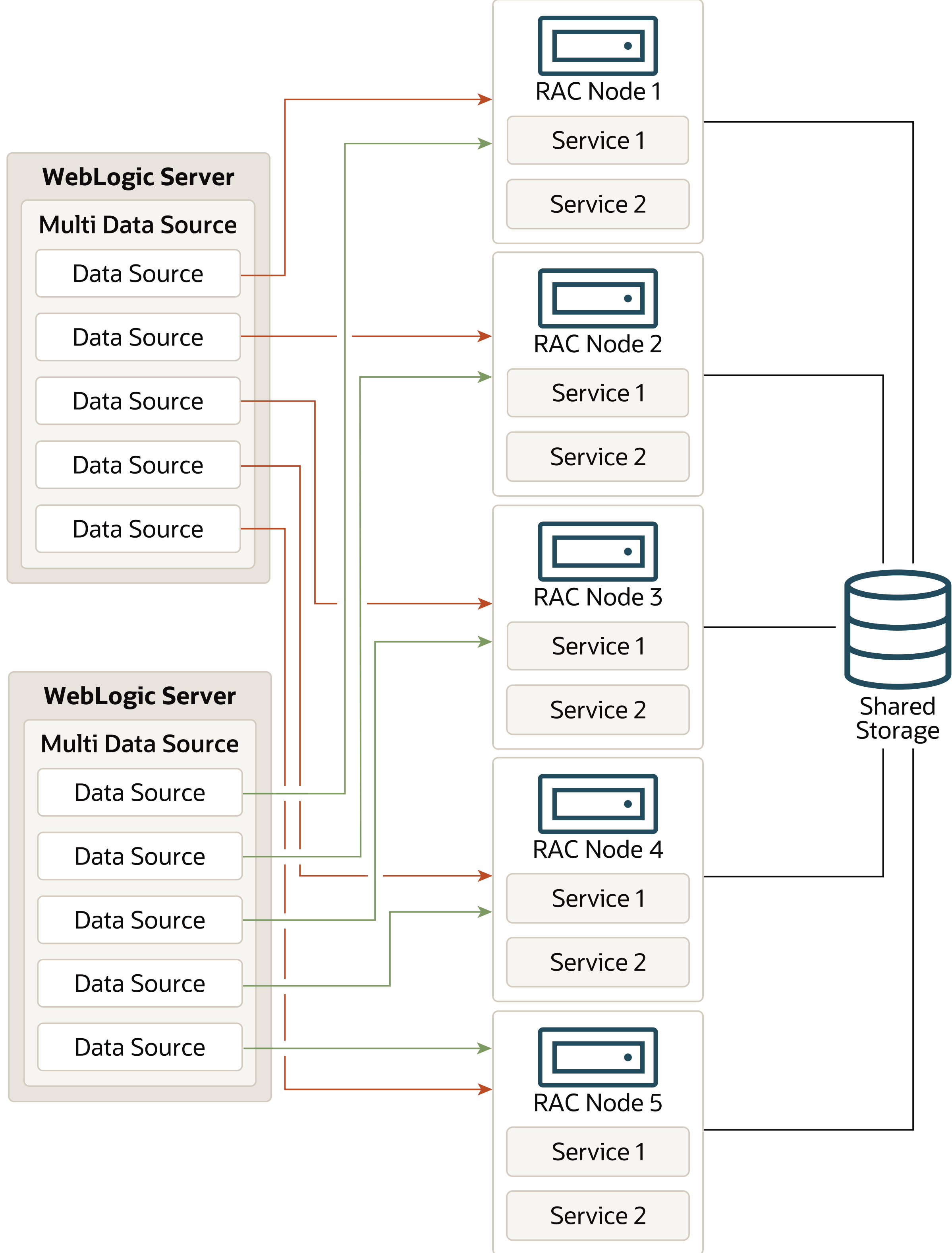
In this example, Service 1 is active on Oracle RAC Nodes 1, 2, and 3, while Service 2 is inactive on those nodes. Service 2 is active on Oracle RAC Nodes 4 and 5, while Service 1 is inactive on those nodes.

If Oracle RAC Node 1 becomes unavailable for any reason, you can start Service 1 on Oracle RAC Node 4. WebLogic Server will detect that the service is running on Node 4, and will begin making connections from the associated backup generic data source to Node 4 as needed.

Load Balancing

In a load balancing configuration, there are multiple services running concurrently on each Oracle RAC node. Each WLS Multi Data Source has an active connection pool configured to connect to a given service on each of the nodes. In this scenario, you would configure each Multi Data Source to use the Load Balancing algorithm.

Figure B-5 Load Balancing with Multi Data Sources



In this example, Service 1 and Service 2 are each actively running on all of the available Oracle RAC nodes. As a result, all of the connection pools in each multi data source will actively make connections in a round-robin fashion, balancing workload among the five nodes.

Connection Pool Capacity Planning

It is important to note the **Maximum Capacity** value you specify for a Generic data source can cause the connection capacity to a given Oracle RAC node to be exceeded. You must consider the following factors when determining how to set this value for each of your Generic data sources:

- The maximum number of simultaneous connections that a Oracle RAC node can support. This is based on the available memory on a given Oracle RAC node and the amount of memory consumed by each service connection (which can vary for each service). Memory consumption by each connection is a major limitation on the amount of work that can be generated from the WLS servers. Exceeding the amount of available memory by creating too many connections from your WLS Generic data sources to a given Oracle RAC node can result in degraded performance on the Oracle RAC node, or could lead to failed connections.

  Available memory for a node should be based on the PGA target attribute (per session memory).
- The *maximum* number of Generic data sources that can potentially create connections to a given Oracle RAC node, and the number of WebLogic server instances to which each Generic data source or Multi Data Source is targeted. For example, if you have one Generic data source that is targeted to three WLS servers, that Generic data source counts as three Generic data sources, as each server uses its own instance of the Generic data source. This is the case whether the servers are part of a cluster or are independent server instances.
- The *maximum* number of services that may be actively running on a given Oracle RAC node, and the memory consumed on the node by each connection to each service.
- The expected relative workload for each service on a given node. For example, the expected workload of Service1 may be double that of the expected workload of Service2.

  Regardless of whether or not a service is always active on a node, you should allocate resources for that service in the event you have to start it on the node.
- Always use the worst-case scenario when setting the **Maximum Capacity** value for your generic data sources. For example, assume that all available services will be actively running on the Oracle RAC node associated with each generic data source.

The following example explains how you could go about determining each Generic data sources **Maximum Capacity** value. Keep in mind that this is a very simple example intended to illustrate the issue conceptually, and that real-world situations are much more complicated. In general, it is best to under-configure yourGeneric data sources with a low **Maximum Capacity** value, monitor your Oracle RAC nodes for memory usage and performance, then adjust the **Maximum Capacity** values upward until you are approaching the maximum capacity of the associated Oracle RAC nodes.

Example

Suppose you have the following basic configuration:

- Five Oracle RAC nodes, each with 16 GB of memory.
- Two services actively running on each Oracle RAC node. Service1 uses 10 MB per connection, Service2 uses 20MB per connection.

- Workload for each service is the same, that is, each service will generate an equivalent number of connections on a given Oracle RAC node.
- Two WebLogic Server clusters. Cluster1 has five servers. Cluster2 has four servers.
- For a given Oracle RAC node, one generic data source is targeted to Cluster1 and is configured to connect to Service1.
- For a given Oracle RAC node, one generic data source is targeted to Cluster 2 and is configured to connect to Service2.

Because Service2 uses twice as much memory per connection as Service1, you should allocate approximately 10 GB of the node's memory for Service 2 and approximately 5GB for Service1.

Because Cluster1 has five WLS servers, there will be five Generic data sources making connection requests to this Oracle RAC node. This gives you 1 GB of memory available for connections from a given generic data source (5GB/5). Each connection requires 10 MB of memory, so the **Maximum Capacity** value for each generic data source targeted to Cluster1 should be 100 or lower.

Because Cluster 2 has four WLS servers, there will be four Generic data sources making connection requests to this Oracle RAC node. This gives you 2.5 GB of memory available for connections from a given generic data source (10GB/4). Each connection requires 20 MB, so the **Maximum Capacity** value for each generic data source targeted to Cluster2 should be 125 or lower.

If Service 2 generates more workload than Service1, you would have to adjust these values appropriately (increase the **Maximum Capacity** value for the generic data source connecting to Service2, decrease the value for the generic data source connecting to Service1). As long as:

```
(Max. connections to Service1 x memory used per connection) + (Max. connections
to Service2 x memory used per connection) < Available memory
```

you can avoid the potential for performance degradation or connection failures.

Alternatively, in a simple configuration, such as is shown in Figure B-6, the **Maximum Capacity** value you specify for each of your generic data sources can be loosely determined using the following formula:

```
Maximum connection pool capacity = Maximum number of connections to Oracle RAC nodes/
(Number of WebLogic Server instances x Nmber of generic data sources targeted to each
instance x Number of active Oracle RAC services configured x Number of Oracle RAC Nodes)
```

where:

Maximum number of connections to Oracle RAC nodes is determined by total memory available on all nodes divided by the memory consumed by each connection.

Number of WebLogic Server instances is the number of server instances to which the Generic data sources are targeted. If the generic data sources is targeted to a WLS cluster, this is the number of servers in the cluster.

In the example in Figure B-6:

- assume that a maximum of 4000 total connections can be made to the group of Oracle RAC nodes, based on 8GB of available memory per Oracle RAC node, and 10 MB of memory used per connection.
- there are a total of five server instances to which the Generic data sources are targeted
- there are five generic data sources targeted to each server instance

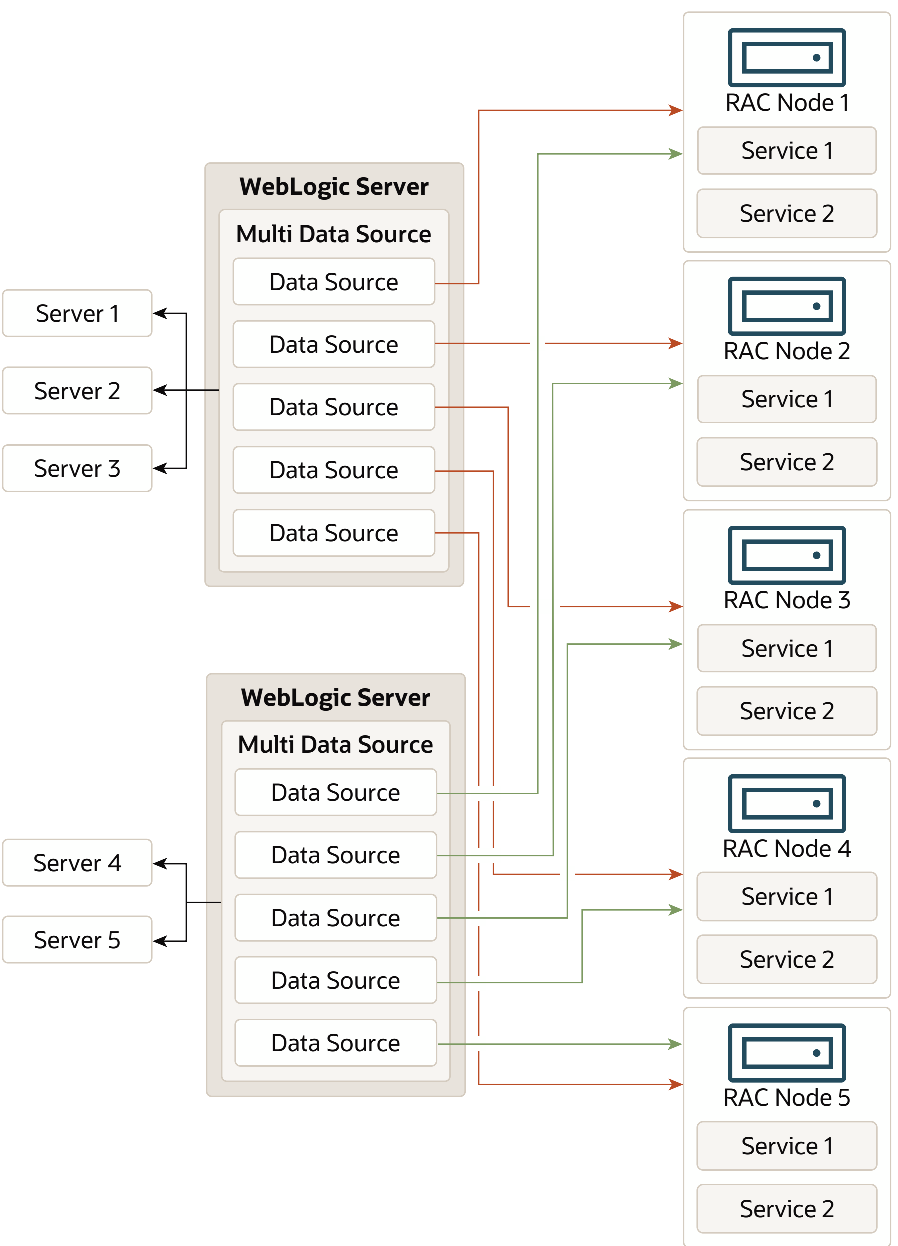
- there are two services running on each Oracle RAC node, and
- there are five Oracle RAC nodes.

In this configuration, the **Maximum Capacity** value you would enter for each of your Generic data sources would be:

```
Maximum connection pool capacity = 4000/(5 server instances x 5 generic data sources x 2
services x 5 Oracle RAC nodes)
```

which would give you a **Maximum Capacity** value of 16 for each of your Generic data sources.

Figure B-6 Example Multi Data Source Connection Configuration



Keep in mind that this formula is just a general guideline for configuring your generic data sources, as many configurations will be too complex for you to use such a simple calculation.

When calculating the **Maximum Capacity** value you should use, always consider the worst-case scenario that you will have in your overall configuration. It is best to under-configure this value for normal operation than to have it over-configured when a worst-case situation develops. You can always monitor your Oracle RAC nodes to determine if it is safe to increase the **Maximum Capacity** value for any of your Generic data sources.

Using SCAN Addresses with Multi Data Sources

Use Single Client Access Name (SCAN) for providing connection to time listener failover and load-balancing.

SCAN is not recommended for use with Multi Data Source. This can be a problem if your configuration is set up to use SCAN (for example, you can't use non-scan addresses if the database listener is set up to use SCAN).

Connection load-balancing cannot be used with a Multi Data Source because the Multi Data Source must be in control of handling the connection load balancing and failover. To turn off this capability, use a URL with an `INSTANCE_NAME` attribute. Each of the Generic data sources in the Multi Data Source should point to a different instance. When the Multi Data Source recognizes that an instance is down on the first Generic data source, it guides connections to the instance on the first Generic data source that is not down. When SCAN used with an `INSTANCE_NAME` attribute, the Multi Data Source provides load-balancing, failover of connections, and continues to provide a more reliable way to get to a listener.

If you need to configure SCAN address for a Multi Data Source, configure each Generic data source member with a URL that has a different `INSTANCE_NAME` value. For example:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=scanname)(PORT=scanport))
(CONNECT_DATA=(SERVICE_NAME=myservice)(INSTANCE_NAME=myinstance)))
```

Note:

If you add a node, you need to manually add a Generic data source member and add it to the Multi Data Source.

Another way to avoid having SCAN do connection load-balancing is to specify a service name that is available on a single instance and omit the instance name. Each member data source must have it's own unique service name and each service name must be available on only one instance that doesn't overlap with any other member data source.

XA Considerations and Limitations when using Multi Data Sources with Oracle RAC

Learn about the certain requirements and limitations you need to consider when using XA (Global transactions) withMulti Data Sources on Oracle RAC.

- Oracle RAC XA Requirements when using Multi Data Sources
- Known Issue Occurring After Database Server Crash

Oracle RAC XA Requirements when using Multi Data Sources

Oracle RAC has the following requirements when using Multi Data Sources with Global transactions:

- Always use a Multi Data Source when using XA transactions with Multi Data Sources for Oracle RAC.
- Global transactions must be initiated, prepared, and concluded in the same instance of the Oracle RAC cluster. WebLogic Server Generic data sources manage this for you when you set `KeepXAConnTillTxComplete="true"` in the Generic data source configuration.
- When using global transactions, transaction IDs (XIDs) must be unique within the Oracle RAC cluster. However, neither the Oracle Thin driver nor an Oracle RAC instance can determine if an XID is unique within the Oracle RAC cluster. Transactions with the same XID can execute SQL code on different instances of the Oracle RAC cluster without any exception.

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.

For a list of the major services that use the JDBC store, see Monitoring Store Connections in *Administering the WebLogic Persistent Store*.

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

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 generic data source that is configured to support global transactions. The JDBC store must use a generic data source that uses a non-XA JDBC driver. For more information about this configuration option, see Using Multi Data Sources without Global Transactions.

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.

In short, for a JDBC store:

- Use a non-XA driver
- Configure the multi data source for Failover mode.

Automatic Retry for JMS Connections

JMS has a limited connection retry mechanism which enables it to silently react to the failure of the Oracle RAC node that hosts its database connection. If the database has experienced either a minor network 'hiccup' or a Oracle RAC database has failed over to another node, the second connection attempt (the retry) will succeed to the next Oracle 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 Oracle 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 either restart WebLogic Server or configure Automatic Service Migration (ASM) `restart-in-place` option for Singleton Services. Otherwise, when the I/O fails, the transaction is marked in a way that cannot be recovered until the JMS server is restarted.

The automatic connection 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 to 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 is only be attempted once. If a failure occurs on the second attempt, an exception is propagated up the call stack and a manual restart is 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. Automatic Service Migration (ASM) `restart-in-place` option for Singleton Services can be used to trigger an automatic restart of failed JMS Services.
>
> The automatic retry delay only applies to the connection retry mechanism. There is no configurable retry delay available for JDBC Store I/O failures.