

Oracle® Fusion Middleware

Configuring and Managing JDBC Data Sources for Oracle
WebLogic Server

11g Release 1 (10.3.6)

E13737-11

April 2014

This document provides JDBC data source configuration and administration information.

Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server, 11g Release 1 (10.3.6)

E13737-11

Copyright © 2007, 2014, Oracle and/or its affiliates. All rights reserved.

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

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

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

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

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

Intel and Intel Xeon 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, Opteron, the AMD logo, and the AMD Opteron 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 on 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. 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.

Contents

Preface	xiii
Documentation Accessibility	xiii
Conventions	xiii
1 Introduction and Roadmap	
1.1 Document Scope and Audience	1-1
1.2 Guide to this Document	1-1
1.3 Related Documentation	1-2
1.4 JDBC Samples and Tutorials	1-3
1.4.1 Avitek Medical Records Application (MedRec) and Tutorials	1-3
1.4.2 JDBC Examples in the WebLogic Server Distribution	1-3
1.5 New and Changed JDBC Data Source Features in This Release	1-3
1.5.1 CapacityIncrement Attribute	1-4
1.5.2 MinCapacity Attribute	1-4
1.5.3 Define Fatal Error Codes	1-4
1.5.4 Data Source Profile Logging	1-4
1.5.5 Application-scoped Drivers	1-4
1.5.6 Oracle BI Server Support	1-5
1.5.7 Keep Connection After Global Transaction	1-5
1.5.8 Session Affinity Policy	1-5
1.5.9 Secure RMI Driver Communication	1-5
1.5.10 Proxy Authentication	1-5
1.5.11 Set Database Credentials on a Connection	1-5
1.5.12 Connection Harvesting	1-5
1.5.13 Connection Labeling	1-5
1.5.14 New Debug Scopes	1-5
1.5.15 ONS Debugging	1-6
2 Configuring WebLogic JDBC Resources	
2.1 Understanding JDBC Resources in WebLogic Server	2-1
2.2 Ownership of Configured JDBC Resources	2-1
2.3 Data Source Configuration Files	2-2
2.3.1 JDBC System Modules	2-2
2.3.2 JDBC Application Modules	2-4
2.3.2.1 Including Drivers in EAR/WAR Files	2-5

2.3.3	JDBC Module File Naming Requirements.....	2-6
2.3.4	JDBC Modules in Versioned Applications.....	2-6
2.3.5	JDBC Schema.....	2-6
2.4	JMX and WLST Access for JDBC Resources.....	2-7
2.4.1	JDBC MBeans for System Resources.....	2-7
2.4.2	JDBC Management Objects in the Java EE Management Model (JSR-77 Support) ...	2-8
2.4.3	Using WLST to Create JDBC System Resources.....	2-9
2.4.4	How to Modify and Monitor JDBC Resources.....	2-10
2.4.5	Best Practices when Using WLST to Configure JDBC Resources.....	2-10
2.5	Creating High-Availability JDBC Resources.....	2-11

3 Configuring JDBC Data Sources

3.1	Understanding JDBC Data Sources.....	3-1
3.2	Types of WebLogic Server JDBC Data Sources.....	3-1
3.3	Creating a JDBC Data Source.....	3-2
3.3.1	JDBC Data Source Properties.....	3-2
3.3.1.1	Data Source Names.....	3-2
3.3.1.2	JNDI Names.....	3-2
3.3.1.3	Selecting a Database Type.....	3-2
3.3.1.4	Selecting a JDBC Driver.....	3-3
3.3.2	Configure Transaction Options.....	3-4
3.3.3	Configure Connection Properties.....	3-5
3.3.3.1	Configuring Connection Properties for Oracle BI Server.....	3-5
3.3.4	Test Connections.....	3-5
3.3.5	Target the Data Source.....	3-6
3.4	Configuring Connection Pool Features.....	3-6
3.4.1	Enabling JDBC Driver-Level Features.....	3-6
3.4.2	Enabling Connection-based System Properties.....	3-6
3.4.3	Initializing Database Connections with SQL Code.....	3-7
3.5	Advanced Connection Properties.....	3-7
3.5.1	Define Fatal Error Codes.....	3-7
3.6	Configuring Oracle Parameters.....	3-8
3.7	Configuring an ONS Client.....	3-8
3.8	Tuning Generic Data Source Connection Pools.....	3-8
3.9	Setting Database Security Credentials.....	3-8
3.9.1	Types of Data Source Pools.....	3-9
3.9.2	Using a User Name/Password.....	3-9
3.9.3	Set Client ID On Connection.....	3-9
3.9.4	Identity-based Connection Pooling.....	3-10
3.9.4.1	How Heterogeneous Connections are Created.....	3-10
3.9.4.2	Using Identity-based Pooling with Global Transactions.....	3-11
3.9.4.3	Using Identity-based Pooling with LLR.....	3-11
3.10	JDBC Data Source Factories (Deprecated).....	3-11

4 Using GridLink Data Sources

4.1	What is a GridLink Data Source.....	4-1
4.1.1	Fast Connection Failover.....	4-2

4.1.2	Runtime Connection Load Balancing	4-3
4.1.3	Graceful Handling for Oracle RAC Outages	4-4
4.1.3.1	Handling for Oracle RAC Outages Prior to Oracle RAC 11.2	4-5
4.1.4	GridLink Affinity	4-5
4.1.4.1	Session Affinity Policy	4-5
4.1.4.2	XA Affinity Policy	4-6
4.1.5	SCAN Addresses	4-7
4.1.6	Secure Communication using Oracle Wallet	4-7
4.2	Creating a GridLink Data Source	4-7
4.2.1	JDBC Data Source Properties	4-8
4.2.1.1	Data Source Names	4-8
4.2.1.2	JNDI Names	4-8
4.2.1.3	Select an XA or Non-XA Driver	4-8
4.2.2	Configure Transaction Options	4-8
4.2.3	Configure Connection Properties	4-9
4.2.3.1	Enter Connection Properties	4-10
4.2.3.2	Enter a Complete URL	4-10
4.2.4	Test Connections	4-10
4.2.5	Configure an ONS Client Configuration	4-10
4.2.5.1	Secure ONS Client Communication	4-11
4.2.6	Test ONS Client Configuration	4-11
4.2.7	Target the Data Source	4-11
4.3	Using Socket Direct Protocol	4-11
4.3.1	Configuring Runtime Load Balancing	4-11
4.4	Configuring Connection Pool Features	4-12
4.4.1	Enabling JDBC Driver-Level Features	4-13
4.4.2	Enabling Connection-based System Properties	4-13
4.4.3	Initializing Database Connections with SQL Code	4-13
4.5	Configuring Oracle Parameters	4-14
4.6	Configuring an ONS Client	4-14
4.6.1	Enabling FAN Events	4-14
4.6.2	Configuring Generic Data Source Connection Testing	4-14
4.6.3	Using a Wallet File	4-14
4.7	Configuring Oracle Parameters	4-14
4.8	Tuning GridLink Data Source Connection Pools	4-14
4.9	Setting Database Security Credentials	4-15
4.9.1	Using a User Name/Password	4-15
4.9.2	Set Client ID On Connection	4-15
4.10	Monitoring GridLink JDBC Resources	4-16
4.10.1	Viewing Run-Time Statistics	4-16
4.10.1.1	JDBCOracleDataSourceInstanceRuntimeMBean	4-16
4.10.1.2	JDBCDataSourceRuntimeMBean	4-16
4.10.1.3	ONSDaemonRuntimeMBean	4-16
4.10.2	Debug GridLink Data Sources	4-16
4.10.2.1	JDBC Debugging Scopes	4-16
4.10.2.2	UCP JDK Logging	4-17
4.10.2.3	Enable Debugging Using the Command Line	4-17

4.11	Best Practices for GridLink Data Sources	4-17
4.11.1	Catch and Handle Exceptions	4-17

5 Configuring JDBC Multi Data Sources

5.1	Multi Data Source Features	5-1
5.1.1	Removing a Database Node	5-2
5.1.2	Adding a Database Node	5-2
5.2	Creating and Configuring Multi Data Sources	5-2
5.3	Choosing the Multi Data Source Algorithm	5-3
5.3.1	Failover	5-3
5.3.2	Load Balancing	5-3
5.4	Multi Data Source Fail-Over Limitations and Requirements	5-3
5.4.1	Test Connections on Reserve to Enable Fail-Over	5-3
5.4.2	No Fail-Over for In-Use Connections	5-4
5.5	Multi Data Source Failover Enhancements	5-4
5.5.1	Connection Request Routing Enhancements When a Data Source Fails	5-4
5.5.2	Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source 5-4	
5.5.3	Enabling Failover for Busy Data Sources in a Multi Data Source	5-5
5.5.4	Controlling Multi Data Source Failover with a Callback	5-5
5.5.4.1	Callback Handler Requirements	5-5
5.5.4.2	Callback Handler Configuration	5-6
5.5.4.3	How It Works—Failover	5-6
5.5.5	Controlling Multi Data Source Failback with a Callback	5-7
5.5.5.1	How It Works—Failback	5-7
5.6	Deploying JDBC Multi Data Sources on Servers and Clusters	5-8

6 Advanced Configurations for Oracle Drivers and Databases

6.1	Options to Improve Driver Performance	6-1
6.2	Proxy Authentication for Oracle Databases	6-1
6.2.1	How to Configure Proxy Authentication	6-1
6.3	Setting Database Credentials on a Connection	6-2
6.3.1	Considerations When Setting Database Credentials on a Connection	6-3
6.4	Configuring Connection Harvesting	6-3
6.4.1	Enable Connection Harvesting	6-4
6.4.2	Make Connections Harvestable	6-4
6.4.3	Recover Harvested Connections	6-4
6.5	Labeling Connections	6-5
6.5.1	Implementing Labeling Callbacks	6-6
6.5.2	Creating a Labeling Callback	6-6
6.5.2.1	Example Labeling Callback	6-7
6.5.3	Registering a Labeling Callback	6-9
6.5.3.1	Removing a Labeling Callback	6-9
6.5.3.2	Applying Connection Labels	6-9
6.5.4	Reserving Labeled Connections	6-10
6.5.5	Checking Unmatched labels	6-10
6.5.6	Removing a Connection Label	6-11

7 JDBC Data Source Transaction Options

7.1	Enabling Support for Global Transactions with a Non-XA JDBC Driver	7-2
7.2	Understanding the Logging Last Resource Transaction Option	7-2
7.2.1	Advantages to Using the Logging Last Resource Optimization	7-3
7.2.2	Enabling the Logging Last Resource Transaction Optimization	7-4
7.2.3	Programming Considerations and Limitations for LLR Data Sources	7-4
7.2.4	Administrative Considerations and Limitations for LLR Data Sources	7-5
7.3	Understanding the Emulate Two-Phase Commit Transaction Option	7-6
7.3.1	Limitations and Risks When Emulating Two-Phase Commit Using a Non-XA Driver ...	7-7
7.3.1.1	Heuristic Completions and Data Inconsistency	7-7
7.3.1.2	Cannot Recover Pending Transactions	7-8
7.3.1.3	Possible Performance Loss with Non-XA Resources in Multi-Server Configurations 7-8	
7.3.1.4	Multiple Non-XA Participants	7-8

8 Using Roles and Policies to Secure JDBC Data Sources

8.1	Setting Security Policies for JDBC Resources	8-1
8.2	Security Roles for JDBC MBeans	8-1
8.2.1	JDBC Domain Configuration MBeans	8-1
8.2.2	JDBC System Module MBeans	8-2

9 Deploying Data Sources on Servers and Clusters

9.1	Deploying Data Sources on Servers and Clusters	9-1
9.2	Minimizing Server Startup Hang Caused By an Unresponsive Database	9-1

10 Tuning Data Source Connection Pools

10.1	Increasing Performance with the Statement Cache	10-1
10.1.1	Statement Cache Algorithms	10-2
10.1.1.1	LRU (Least Recently Used)	10-2
10.1.1.2	Fixed	10-2
10.1.2	Statement Cache Size	10-2
10.1.3	Usage Restrictions for the Statement Cache	10-2
10.1.3.1	Calling a Stored Statement After a Database Change May Cause Errors	10-3
10.1.3.2	Using setNull In a Prepared Statement	10-3
10.1.3.3	Statements in the Cache May Reserve Database Cursors	10-3
10.2	Connection Testing Options for a Data Source	10-3
10.2.1	Database Connection Testing Semantics	10-4
10.2.1.1	Connection Testing When Database Connections are Created	10-5
10.2.1.2	Periodic Connection Testing	10-5
10.2.1.3	Testing Reserved Connections	10-6
10.2.1.4	Minimized Connection Test Delay After Database Connectivity Loss	10-6
10.2.1.5	Minimized Connection Request Delay After Connection Test Failures	10-6
10.2.1.6	Minimized Connection Request Delays After Loss of DBMS Connectivity	10-7
10.2.1.7	Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection 10-7	

10.2.2	Database Connection Testing Configuration Recommendations.....	10-7
10.2.3	Default Test Table Name	10-8
10.3	Enabling Connection Creation Retries.....	10-9
10.4	Enabling Connection Requests to Wait for a Connection	10-9
10.4.1	Connection Reserve Timeout.....	10-9
10.4.2	Limiting the Number of Waiting Connection Requests.....	10-10
10.5	Automatically Recovering Leaked Connections	10-10
10.6	Avoiding Server Lockup with the Correct Number of Connections	10-10
10.7	Limiting Statement Processing Time with Statement Timeout.....	10-11
10.8	Using Pinned-To-Thread Property to Increase Performance	10-11
10.8.1	Changes to Connection Pool Administration Operations When PinnedToThread is Enabled 10-11	
10.8.2	Additional Database Resource Costs When PinnedToThread is Enabled	10-12
10.9	Using Unwrapped Data Type Objects	10-12
10.9.1	How to Disable Wrapping.....	10-13
10.9.1.1	Disable Wrapping using the Administration Console.....	10-13
10.9.1.2	Disable Wrapping using WLST	10-13

11 Using WebLogic Server with Oracle RAC

11.1	Overview of Oracle Real Application Clusters	11-1
11.2	Software Requirements	11-2
11.3	JDBC Driver Requirements	11-2
11.4	Hardware Requirements.....	11-2
11.4.1	WebLogic Server Cluster	11-2
11.4.2	Oracle RAC Cluster	11-2
11.4.3	Shared Storage.....	11-2
11.5	Configuration Options in WebLogic Server with Oracle RAC	11-3
11.5.1	Choosing a WebLogic Server Configuration for Use with Oracle RAC.....	11-3
11.5.2	Validating Connections when using WebLogic Server with Oracle RAC	11-4
11.5.3	Additional Considerations When Using WebLogic Server with Oracle RAC	11-4

12 Using JDBC Drivers with WebLogic Server

12.1	JDBC Driver Support.....	12-1
12.2	JDBC Drivers Installed with WebLogic Server.....	12-2
12.3	Setting the Environment for a Third-Party JDBC Driver	12-2
12.4	Globalization Support for the Oracle 11g Thin Driver.....	12-2
12.5	Using the Oracle Thin Driver in Debug Mode	12-3
12.6	Using the Oracle Thin Driver Over SSL	12-3

13 Monitoring WebLogic JDBC Resources

13.1	Viewing Run-Time Statistics	13-1
13.1.1	Data Source Statistics	13-1
13.1.2	Prepared Statement Cache Statistics	13-2
13.2	Profile Logging.....	13-2
13.3	Collecting Profile Information	13-2
13.3.1	Profile Types.....	13-2

13.3.1.1	Connection Usage (PROFILE_TYPE_CONN_USAGE_STR).....	13-3
13.3.1.2	Connection Reservation Wait (PROFILE_TYPE_CONN_RESV_WAIT_STR) .	13-3
13.3.1.3	Connection Reservation Failed (PROFILE_TYPE_CONN_RESV_FAIL_STR)	13-3
13.3.1.4	Connection Leak (PROFILE_TYPE_CONN_LEAK_STR)	13-4
13.3.1.5	Connection Last Usage (PROFILE_TYPE_CONN_LAST_USAGE_STR)	13-4
13.3.1.6	Connection Multithreaded Usage (PROFILE_TYPE_CONN_MT_USAGE_STR).....	13-4
13.3.1.7	Statement Cache Entry (PROFILE_TYPE_STMT_CACHE_ENTRY_STR)	13-5
13.3.1.8	Statements Usage (PROFILE_TYPE_STMT_USAGE_STR).....	13-5
13.3.1.9	Connection Unwrap (WEBLOGIC.JDBC.CONN.UNWRAP).....	13-5
13.3.1.10	Example Profile Information Record Log	13-5
13.3.2	Accessing Diagnostic Data	13-6
13.3.3	Callbacks for Monitoring Driver-Level Statistics (Deprecated).....	13-6
13.4	Debugging JDBC Data Sources.....	13-6
13.4.1	Enabling Debugging.....	13-7
13.4.1.1	Enable Debugging Using the Command Line	13-7
13.4.1.2	Enable Debugging Using the WebLogic Server Administration Console	13-7
13.4.1.3	Enable Debugging Using the WebLogic Scripting Tool	13-7
13.4.1.4	Changes to the config.xml File	13-8
13.4.2	JDBC Debugging Scopes.....	13-9
13.4.3	Setting Debugging for UCP/ONS.....	13-9
13.4.3.1	Debugging UCP	13-9
13.4.3.2	Debugging ONS.....	13-9
13.4.4	Request Dyeing	13-10

14 Managing WebLogic JDBC Resources

14.1	Testing Data Sources and Database Connections	14-1
14.2	Managing the Statement Cache for a Data Source	14-2
14.2.1	Clearing the Statement Cache for a Data Source	14-2
14.2.2	Clearing the Statement Cache for a Single Connection.....	14-2
14.3	Shrinking a Connection Pool.....	14-3
14.4	Resetting a Connection Pool.....	14-3
14.5	Suspending a Connection Pool	14-3
14.6	Resuming a Connection Pool	14-3
14.7	Shutting Down a Data Source	14-4
14.8	Starting a Data Source	14-4
14.9	Managing DBMS Network Failures	14-4

A Configuring JDBC Application Modules for Deployment

A.1	Packaging a JDBC Module with an Enterprise Application: Main Steps	A-1
A.2	Creating Packaged JDBC Modules	A-2
A.2.1	Creating a JDBC Data Source Module Using the Administration Console	A-2
A.2.2	JDBC Packaged Module Requirements	A-2
A.2.3	JDBC Application Module Limitations	A-3
A.2.4	Creating a JDBC Data Source Module	A-3
A.2.5	Creating a JDBC Multi Data Source Module	A-4

A.2.6	Encrypting Database Passwords in a JDBC Module	A-5
A.2.6.1	Deploying JDBC Modules to New Domains	A-5
A.2.7	Application Scoping for a Packaged JDBC Module	A-5
A.3	Referencing a JDBC Module in Java EE Descriptor Files	A-6
A.3.1	Packaged JDBC Module References in weblogic-application.xml	A-6
A.3.2	Packaged JDBC Module References in Other Descriptors	A-7
A.4	Packaging an Enterprise Application with a JDBC Module.....	A-7
A.5	Deploying an Enterprise Application with a JDBC Module	A-8
A.6	Getting a Database Connection from a Packaged JDBC Module	A-8

B WebLogic Server 10.3.6 Support for Oracle 12c Database

B.1	Using an Oracle 12c Database with WebLogic Server 10.3.6.....	B-1
B.1.1	Using 11g Drivers with Oracle 12c Database.....	B-1
B.1.2	Using 12c Drivers with Oracle 12c Database	B-2
B.1.2.1	Required Oracle 12c Driver Files.....	B-2
B.1.2.2	Download Oracle 12c Database Files.....	B-3
B.1.2.3	Update the WebLogic Server CLASSPATH	B-3
B.1.2.4	Other Considerations	B-4
B.2	JDBC 4.1 Support for JDK 7.....	B-5
B.3	Using Application Continuity.....	B-5
B.3.1	How Application Continuity Works	B-6
B.3.2	Requirements and Considerations	B-6
B.3.3	Configuring Application Continuity	B-7
B.3.3.1	Selecting the Driver for Application Continuity.....	B-7
B.3.3.2	Using a Connection Callback.....	B-7
B.3.3.2.1	Create an Initialization Callback.....	B-7
B.3.3.2.2	Registering an Initialization Callback	B-8
B.3.3.2.3	Unregister an Initialization Callback.....	B-8
B.3.3.3	Disabling Application Continuity for a Connection	B-8
B.3.3.4	Configuring Logging for Application Continuity	B-9
B.4	Global Database Services	B-9
B.4.1	Requirements and Considerations	B-9
B.4.2	Creating a GridLink DataSource for GDS Connectivity	B-10
B.5	Container Database with Pluggable Databases.....	B-10
B.5.1	Creating Service for PDB Access	B-10
B.5.2	Setting the PDB using JDBC.....	B-11
B.6	Automatic ONS Listener Support	B-11
B.7	Database Resident Connection Pooling Support	B-11

C Using Multi Data Sources with Oracle RAC

C.1	Overview of Oracle Real Application Clusters	C-1
C.1.1	Oracle RAC Scalability with WebLogic Server Multi Data Sources.....	C-2
C.1.2	Oracle RAC Availability with WebLogic Server Multi Data Sources.....	C-3
C.1.3	Oracle RAC Load Balancing with WebLogic Server Multi Data Sources	C-3
C.2	Software Requirements	C-3
C.3	JDBC Driver Requirements	C-3
C.4	Hardware Requirements.....	C-3

C.4.1	WebLogic Server Cluster	C-3
C.4.2	Oracle RAC Cluster	C-4
C.4.3	Shared Storage.....	C-4
C.5	Configuring Multi Data Sources with Oracle RAC	C-4
C.5.1	Choosing a Multi Data Source Configuration for Use with Oracle RAC.....	C-4
C.5.2	Configuring Multi Data Sources for use with Oracle RAC	C-4
C.5.2.1	Attributes of a Multi Data Source	C-6
C.5.3	Configuration Considerations for Failover.....	C-6
C.5.3.1	Multi Data Source-Managed Failover and Load Balancing.....	C-6
C.5.3.2	Delays During Failover.....	C-6
C.5.3.3	Failure Handling Walkthrough for Global Transactions	C-8
C.5.4	Configuring the Listener Process for Each Oracle RAC Instance.....	C-8
C.5.5	Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled	C-10
C.5.6	Additional Configuration Considerations	C-10
C.6	Using Multi Data Sources with Global Transactions.....	C-11
C.6.1	Rules for Data Sources within a Multi Data Source Using Global Transactions.....	C-11
C.6.2	Required Attributes of Data Sources within a Multi Data Source Using Global Transactions	C-12
C.6.3	Sample Configuration Code.....	C-12
C.7	Using Multi Data Sources without Global Transactions.....	C-14
C.7.1	Attributes of Data Sources within a Multi Data Source Not Using Global Transactions .	C-14
C.7.2	Sample Configuration Code.....	C-14
C.8	Configuring Connections to Services on Oracle RAC Nodes	C-16
C.8.1	Configuring a Data Source to Connect to a Service.....	C-16
C.8.2	Service Connection Configurations	C-17
C.8.2.1	Workload Management.....	C-17
C.8.2.2	Load Balancing.....	C-18
C.8.3	Connection Pool Capacity Planning	C-19
C.9	XA Considerations and Limitations when using multi Data Sources with Oracle RAC	C-22
C.9.1	Oracle RAC XA Requirements when using multi Data Sources	C-23
C.9.1.1	Use Multi Data Sources	C-23
C.9.1.2	A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the Oracle RAC Cluster	C-23
C.9.1.3	Transaction IDs Must Be Unique Within the Oracle RAC Cluster	C-23
C.9.2	Known Limitations When Using Oracle RAC with multi Data Sources.....	C-23
C.9.2.1	Potential for Data Deadlocks in Some Failure Scenarios.....	C-23
C.9.2.2	Potential for Transactions Completed Out of Sequence	C-24
C.9.3	Known Issue Occurring After Database Server Crash.....	C-24
C.10	JDBC Store Recovery with Oracle RAC.....	C-24
C.10.1	Configuring a JDBC Store for Use with Oracle RAC.....	C-24
C.10.2	Automatic Retry	C-25

D Using Connect-Time Failover with Oracle RAC (Deprecated)

D.1	Using Connect-Time Failover without Global Transactions	D-1
-----	---	-----

D.2	Attributes of a Connect-Time Failover Configuration without Global Transactions	D-3
D.3	Sample Configuration Code	D-3

E Using Fast Connection Failover with Oracle RAC

E.1	JDBC Driver Configuration for use with Oracle Fast Connection Failover	E-1
-----	--	-----

Preface

This preface describes the document accessibility features and conventions used in this guide—*Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.

Documentation Accessibility

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

Access to Oracle Support

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

Conventions

The following text conventions are used in this document:

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

Introduction and Roadmap

This section describes the contents and organization of this guide—*Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.

- [Section 1.1, "Document Scope and Audience"](#)
- [Section 1.2, "Guide to this Document"](#)
- [Section 1.3, "Related Documentation"](#)
- [Section 1.4, "JDBC Samples and Tutorials"](#)
- [Section 1.5, "New and Changed JDBC Data Source Features in This Release"](#)

1.1 Document Scope and Audience

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

This document does not address specific JDBC programming topics. For links to WebLogic Server documentation and resources for this topic, see [Section 1.3, "Related Documentation."](#)

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

1.2 Guide to this Document

- This chapter, [Chapter 1, "Introduction and Roadmap,"](#) introduces the organization of this guide and lists new features in the current release.
- [Chapter 2, "Configuring WebLogic JDBC Resources,"](#) provides an overview of WebLogic JDBC resources.
- [Chapter 3, "Configuring JDBC Data Sources,"](#) describes WebLogic JDBC data source configuration.
- [Chapter 4, "Using GridLink Data Sources,"](#) describes WebLogic JDBC GridLink data source configuration.
- [Chapter 5, "Configuring JDBC Multi Data Sources,"](#) describes WebLogic JDBC multi data source configuration.

- [Chapter 6, "Advanced Configurations for Oracle Drivers and Databases,"](#) provides advanced configuration options that can provide improved data source and driver performance when using Oracle drivers and databases.
- [Chapter 7, "JDBC Data Source Transaction Options,"](#) provides information on XA, non-XA, and Global Transaction options for WebLogic data sources.
- [Chapter 8, "Using Roles and Policies to Secure JDBC Data Sources,"](#) provides information on how WebLogic Server uses roles and policies to secure JDBC data sources.
- [Chapter 9, "Deploying Data Sources on Servers and Clusters,"](#) provides information on how to deploy data sources on servers and clusters.
- [Chapter 10, "Tuning Data Source Connection Pools,"](#) provides information on how to properly tune the connection pool attributes in JDBC data sources in your WebLogic Server domain to improve application and system performance.
- [Chapter 11, "Using WebLogic Server with Oracle RAC,"](#) describes how to configure WebLogic Server for use with Oracle Real Application Clusters.
- [Chapter 12, "Using JDBC Drivers with WebLogic Server,"](#) describes how to use JDBC drivers from other sources in your WebLogic JDBC data source configuration.
- [Chapter 13, "Monitoring WebLogic JDBC Resources,"](#) describes how to monitor JDBC resources, gather profile information about database connection usage, and enable JDBC debugging.
- [Chapter 14, "Managing WebLogic JDBC Resources,"](#) describes how to administer data sources.
- [Appendix A, "Configuring JDBC Application Modules for Deployment,"](#) describes how to configure generic data sources for use with Oracle Real Application Clusters.
- [Appendix C, "Using Multi Data Sources with Oracle RAC,"](#) provides information on how to configure and use multi data sources when using Oracle Real Application Clusters (RAC) with WebLogic Server.
- [Appendix D, "Using Connect-Time Failover with Oracle RAC \(Deprecated\),"](#) provides information on how WebLogic Server provides Connect-Time Failover (deprecated) for legacy applications that use data sources configured to use connect-time failover and load balancing.
- [Appendix E, "Using Fast Connection Failover with Oracle RAC,"](#) provides information on how to use WebLogic server with Oracle Fast Connection Failover.

1.3 Related Documentation

This document contains JDBC data source configuration and administration information.

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

- *Programming JDBC 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.

1.4 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.

1.4.1 Avitek Medical Records Application (MedRec) and Tutorials

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

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

1.4.2 JDBC Examples in the WebLogic Server Distribution

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

1.5 New and Changed JDBC Data Source Features in This Release

This release includes the following new and changed features:

- [Section 1.5.1, "CapacityIncrement Attribute"](#)
- [Section 1.5.2, "MinCapacity Attribute"](#)
- [Section 1.5.3, "Define Fatal Error Codes"](#)
- [Section 1.5.4, "Data Source Profile Logging"](#)
- [Section 1.5.5, "Application-scoped Drivers"](#)
- [Section 1.5.6, "Oracle BI Server Support"](#)
- [Section 1.5.7, "Keep Connection After Global Transaction"](#)
- [Section 1.5.8, "Session Affinity Policy"](#)
- [Section 1.5.9, "Secure RMI Driver Communication"](#)
- [Section 1.5.10, "Proxy Authentication"](#)
- [Section 1.5.11, "Set Database Credentials on a Connection"](#)
- [Section 1.5.12, "Connection Harvesting"](#)
- [Section 1.5.13, "Connection Labeling"](#)

- [Section 1.5.14, "New Debug Scopes"](#)
- [Section 1.5.15, "ONS Debugging"](#)

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

1.5.1 CapacityIncrement Attribute

In WebLogic Server 10.3.1 and higher releases, the `capacityIncrement` attribute is no longer configurable and is set to a value of 1.

1.5.2 MinCapacity Attribute

The `MinCapacity` attribute sets the minimum number of physical connections that a connection pool can contain after it is initialized.

The `InitialCapacity` value that previously handled both the initial and minimum capacity for the pool this has been split into two attributes:

- `MinCapacity` defaults to `InitialCapacity` if not set; `InitialCapacity` continues to default to 1.
- `MinCapacity` is only used for shrinking calculations only. It is lazy in that the minimum connections are not created when the server starts up; `InitialCapacity` is used for this function.
- For upward compatibility, `InitialCapacity` is used if `MinCapacity` is not set.
- After a data source goes through a suspend/resume, the greater of `MinCapacity` or `InitialCapacity` is used.
- Example: You set `InitialCapacity` to 0 to start up database and then set `InitialCapacity` to 50 to keep at least 50 connections in the pool at all times. With the rules above, `MinCapacity` is initially be treated as 0 and then treated as 50 so there will be at least 50 connections in the pool at all times.

See [Section 14.3, "Shrinking a Connection Pool."](#)

1.5.3 Define Fatal Error Codes

For each data source, you can define fatal error codes that indicate that the back-end database with which the data source communicates is no longer accessible on a connection. See [Section 3.5.1, "Define Fatal Error Codes."](#)

1.5.4 Data Source Profile Logging

Prior to WebLogic Server 10.3.6, data source profile records were recorded as WLDF events. To provide better usability and performance, WebLogic Server now uses a data source profile log to store events. See [Section 13, "Monitoring WebLogic JDBC Resources."](#)

1.5.5 Application-scoped Drivers

You can include a database driver in the EAR/WAR file that contains an application-scoped data source. This allows you to deploy a self-contained EAR file that has both the data source and driver required for an application. See [Section 2.3.2.1, "Including Drivers in EAR/WAR Files."](#)

1.5.6 Oracle BI Server Support

WebLogic Server 10.3.6.0 and higher supports the Oracle BI Server. Select **Oracle BI Server** as the database type when creating a new generic data source to interoperate with the Oracle BI Server. See [Section 3.3, "Creating a JDBC Data Source."](#)

1.5.7 Keep Connection After Global Transaction

This release provides a new attribute, `Keep Connection After Global Transaction`, that enables WebLogic Server to keep a physical database connection associated with a logical connection when committing or rolling back a global transaction. See `KeepConnAfterGlobalTx` in the *Oracle WebLogic Server MBean Reference*.

1.5.8 Session Affinity Policy

WebLogic Server GridLink data sources use the session affinity policy to improve performance by directing the database operations of a servlet session to the same RAC instance in a RAC cluster. See [Section 4.1.4, "GridLink Affinity."](#)

1.5.9 Secure RMI Driver Communication

Secure RMI driver client communication with `DataSource` objects at the server level by setting the `RMI JDBC Security` parameter, see "Security Considerations for WebLogic RMI Drivers" in *Programming JDBC for Oracle WebLogic Server*.

1.5.10 Proxy Authentication

You can configure a WebLogic data source to allow a client to connect to an Oracle database through an application server as a proxy user. See [Section 6.2, "Proxy Authentication for Oracle Databases."](#)

1.5.11 Set Database Credentials on a Connection

You can configure a WebLogic data source to allow a connection to set database credentials when connecting to an Oracle database. See [Section 6.3, "Setting Database Credentials on a Connection."](#)

1.5.12 Connection Harvesting

You can specify a number of reserved connections to be released when a data source reaches a specified number of available connections. Harvesting helps to ensure that a specified number of connections are always available in the pool and improves performance by minimizing connection initialization. See [Section 6.4, "Configuring Connection Harvesting."](#)

1.5.13 Connection Labeling

Labeling allows an application to attach arbitrary name/value pairs (labels) to a connection that has a particular initialization state. This allows the application to improve performance by minimizing the time and cost of re-initializing a connection. See [Section 6.5, "Labeling Connections."](#)

1.5.14 New Debug Scopes

This release provides the following new debug scopes:

- `weblogic.jdbc.rac.DebugJDBCONS` - low-level ONS debugging
- `weblogic.jdbc.rac.DebugJDBCRCRAC` - RAC debugging
- `weblogic.jdbc.rac.DebugJDBCUCP` - low-level UCP debugging
- `weblogic.jdbc.rac.DebugJDBCREPLAY` - REPLAY debugging
- `weblogic.jdbc.transaction.DebugJTAJDBC` - transaction debugging

See [Section 13.4.2, "JDBC Debugging Scopes."](#)

1.5.15 ONS Debugging

For WebLogic Server releases 10.3.6.0 and higher, the package names for UCP and ONS are no longer repackaged. This affects debugging for these components, see [Section 13.4.3, "Setting Debugging for UCP/ONS."](#)

Configuring WebLogic JDBC Resources

This chapter describes WebLogic JDBC resources, how they are configured, and how those resources apply to a WebLogic domain.

- [Section 2.1, "Understanding JDBC Resources in WebLogic Server"](#)
- [Section 2.2, "Ownership of Configured JDBC Resources"](#)
- [Section 2.3, "Data Source Configuration Files"](#)
- [Section 2.4, "JMX and WLST Access for JDBC Resources"](#)
- [Section 2.5, "Creating High-Availability JDBC Resources"](#)

2.1 Understanding JDBC Resources in WebLogic Server

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

Each data source that you configure contains a pool of database connections that are created when the data source instance is created—when it is deployed or targeted, or at server startup. Applications lookup a data source on the JNDI tree or in the local application context (`java:comp/env`), depending on how you configure and deploy the object, and then request a database connection. When finished with the connection, the application calls `connection.close()`, which returns the connection to the connection pool in the data source. For more information about data sources in WebLogic Server, see [Section 3, "Configuring JDBC Data Sources"](#) and [Section 4, "Using GridLink Data Sources."](#)

A multi data source is an abstraction around a selected list of data sources that provides load balancing or failover processing between the data sources associated with the multi data source. Multi data sources are bound to the JNDI tree or local application context just like data sources are bound to the JNDI tree. Applications lookup a multi data source on the JNDI tree or in the local application context (`java:comp/env`) just like they do for data sources, and then request a database connection. The multi data source determines which data source to use to satisfy the request depending on the algorithm selected in the multi data source configuration: load balancing or failover. For more information about multi data sources, see [Section 5, "Configuring JDBC Multi Data Sources."](#)

2.2 Ownership of Configured JDBC Resources

A key to understanding WebLogic JDBC data source configuration and management is that *who* creates a JDBC resource or *how* a JDBC resource is created determines how a

resource is deployed and modified. Both WebLogic Administrators and programmers can create JDBC resources:

- WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JDBC modules. These JDBC modules are considered *system modules*. See [Section 2.3.1, "JDBC System Modules"](#) for more details.
- Programmers create modules in a development tool that supports creating an XML descriptor file, then package the JDBC modules with an application and pass the application to a WebLogic Administrator to deploy. These JDBC modules are considered *application modules*. See [Section 2.3.2, "JDBC Application Modules"](#) for more details.

[Table 2-1](#) lists the JDBC module types and how they can be configured and modified.

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

Module Type	Created with	Add/Remove Modules with Administration Console	Modify with JMX (remotely)	Modify with JSR-88 (non-remotely)	Modify with Administration Console
System	Administration Console or WLST	Yes	Yes	No	Yes—via JMX
Application	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

2.3 Data Source Configuration Files

WebLogic JDBC data source configurations are stored in XML documents that conform to the `jdbc-data-source.xsd` schema (available at <http://xmlns.oracle.com/weblogic/jdbc-data-source/1.2/jdbc-data-source.xsd>).

You create and manage JDBC resources either as system modules or as application modules. JDBC application modules are a WebLogic-specific extension of Java EE modules and can be configured either within a Java EE application or as stand-alone modules.

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

2.3.1 JDBC System Modules

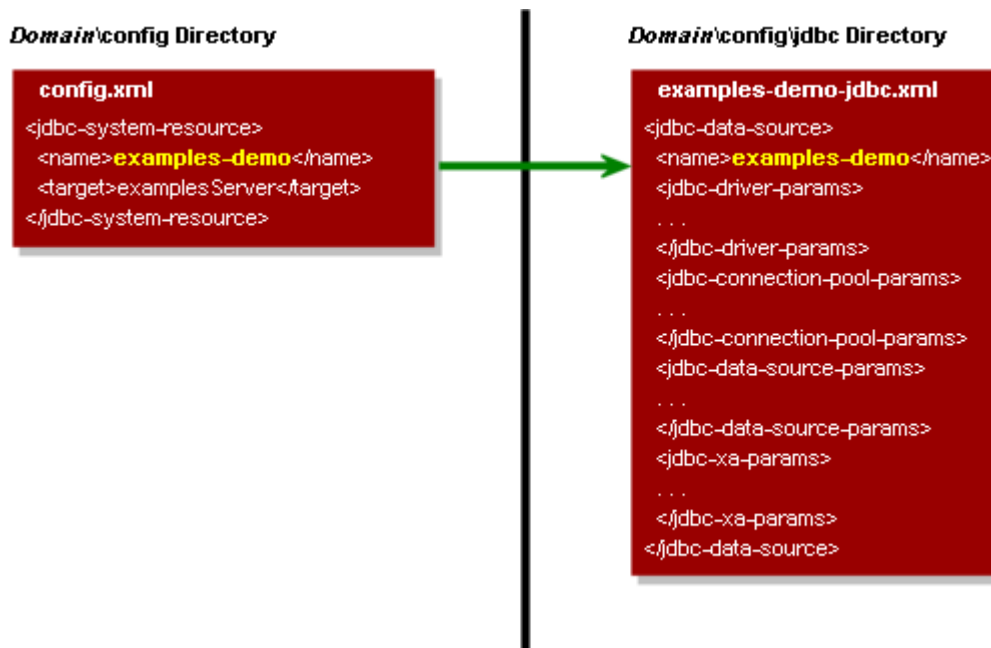
When you create a JDBC resource (data source or multi data source) using the Administration Console or using the WebLogic Scripting Tool (WLST), WebLogic Server creates a JDBC module in the `config/jdbc` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file. The JDBC module conforms to the `jdbc-data-source.xsd` schema (available at <http://xmlns.oracle.com/weblogic/jdbc-data-source/1.2/jdbc-data-source.xsd>).

JDBC resources that you configure this way are considered *system modules*. System modules are owned by an Administrator, who can delete, modify, or add similar resources at any time. System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications

deployed on the same targets and to client applications. System modules are also accessible through JMX as `JDBCSystemResourceMBeans`.

Data source system modules are included in the domain's `config.xml` file as a `JDBCSystemResource` element, which includes the name of the JDBC module file and the list of target servers and clusters on which the module is deployed. [Figure 2-1](#) shows an example of a data source listing in a `config.xml` file and the module that it maps to.

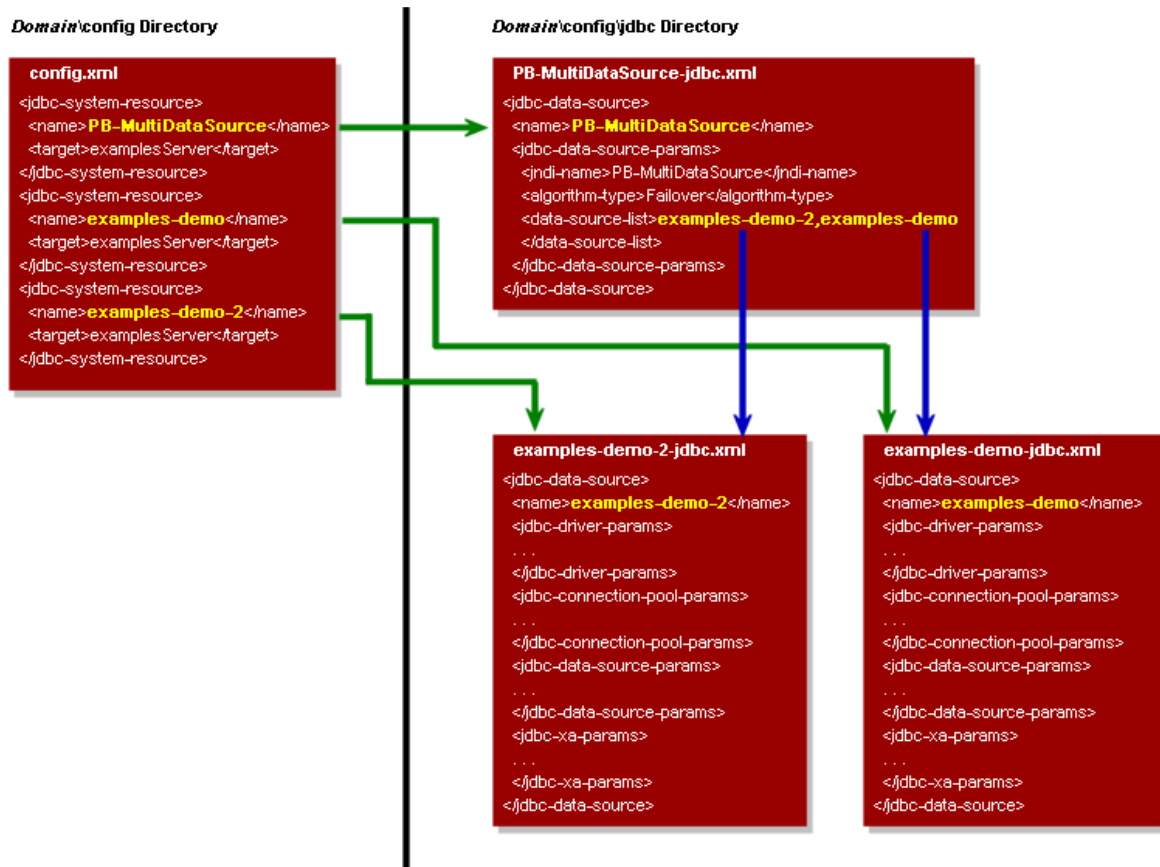
Figure 2-1 Reference from config.xml to a Data Source System Module



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.

Similarly, multi data source system modules are included in the domain's `config.xml` file as a `jdbc-system-resource` element. The multi data source module includes a `data-source-list` parameter that maps to the data source modules used by the multi data source. The individual data source modules are also included in the `config.xml` file. [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 multi data source and the two data sources used by the multi data source, which are also listed within the multi data source module. Your application can look up any of these modules on the JNDI tree and request a database connection. If you look up the multi data source, the multi data source determines which of the other data sources to use to supply the database connection, depending on the data sources in the `data-source-list` parameter, the order in which the data sources are listed, and the algorithm specified in the `algorithm-type` parameter. For more information about multi data sources, see [Section 5, "Configuring JDBC Multi Data Sources."](#)

2.3.2 JDBC Application Modules

JDBC resources can also be managed as application modules, similar to standard Java EE modules. A JDBC application module is simply an XML file that conforms to the `jdbc-data-source.xsd` schema (available at <http://xmlns.oracle.com/weblogic/jdbc-data-source/1.2/jdbc-data-source.xsd>) and represents a data source or a multi data source.

JDBC modules can be included as part of an Enterprise Application as a *packaged module*. Packaged modules are bundled with an EAR or exploded EAR directory, and are referenced in all appropriate deployment descriptors, such as the `weblogic-application.xml` and `ejb-jar.xml` deployment descriptors. The JDBC module is deployed along with the enterprise application, and can be configured to be available only to the enclosing application or to all applications. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments. With

packaged JDBC modules, you can migrate your application and the required JDBC configuration from environment to environment, such as from a testing environment to a production environment, without opening an EAR file and without extensive manual data source reconfiguration.

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

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

A JDBC application module can also be deployed as a stand-alone resource using the `weblogic.Deployer` utility or the Administration Console, in which case the resource is typically available to the server or cluster targeted during the deployment process. JDBC resources deployed in this manner are called *stand-alone modules* and can be reconfigured using the Administration Console or a JSR-88 compliant tool, but are unavailable through JMX or WLST.

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

Note: When deploying 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 data sources.

For more information about JDBC application modules, see [Section A, "Configuring JDBC Application Modules for Deployment."](#)

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

2.3.2.1 Including Drivers in EAR/WAR Files

In WebLogic Server 10.3.6 and higher releases, you can include a database driver in the `APP-INF/lib` directory of the EAR/WAR file that contains an application-scoped 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 `web.xml` file to ensure the application uses the driver classes that it was packaged with.

2.3.3 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.

2.3.4 JDBC Modules in Versioned Applications

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

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

This name is used for data source 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" in *Developing Applications for Oracle WebLogic Server*
- "Using Production Redeployment to Update Applications" in *Deploying Applications to Oracle WebLogic Server*

2.3.5 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://xmlns.oracle.com/weblogic/jdbc-data-source/1.2/jdbc-data-source.xsd>.

Note: The *scope* in the `jdbc-data-source-params` element of the schema may only be set to `Application` for application-scoped packaged data sources. The value `Application` is not valid for:

- System resources in `config/jdbc`, including generic, multi-data sources, and GridLink data sources.
- Stand-alone data sources that are deployed dynamically or statically using the `<app-deployment>` element in the `config.xml` file.

For these data source types, there is no application to scope the data source and no associated module. WebLogic Server does not generate a scope of `Application`. This omission was not flagged as an error in releases of prior to WebLogic Server 10.3.6.0 and is displayed in the console with an invalid name similar to `ds0@null@ds0`. For WebLogic Server 10.3.6.0 and higher, an `Error` message is logged for this configuration error and the system attempts to set the scope to `Global` and display the data source name as `ds0`. In future releases, this error may be treated as fatal.

2.4 JMX and WLST Access for JDBC Resources

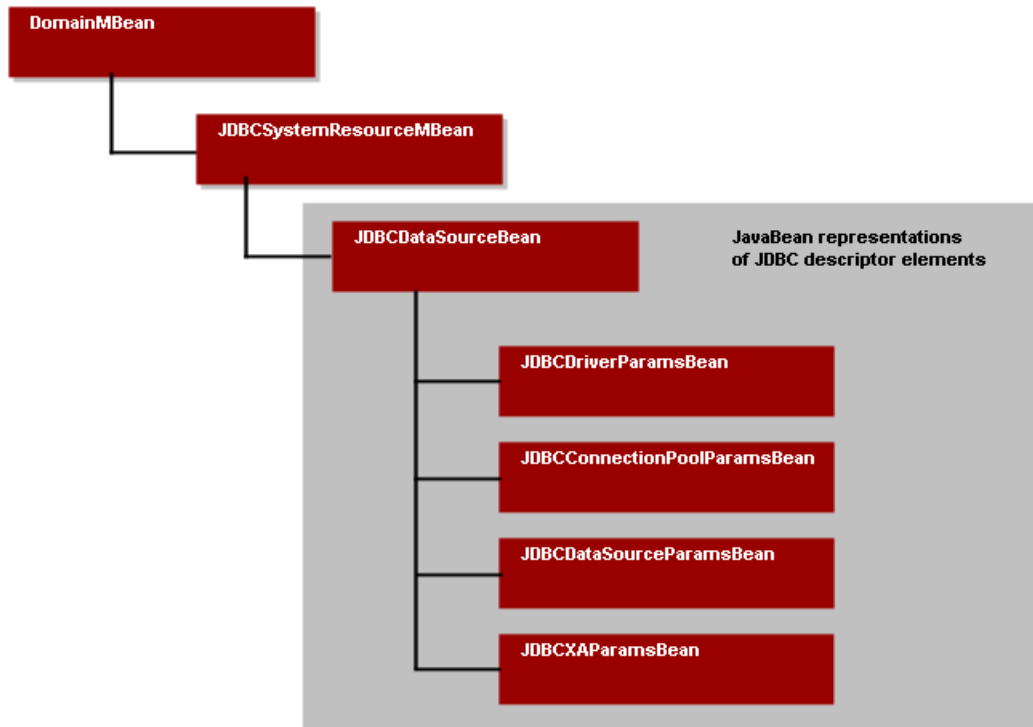
When you create JDBC resources using the Administration Console or WLST, WebLogic Server creates MBeans (Managed Beans) for each of the resources. You can then access these MBeans using JMX or the WebLogic Scripting Tool (WLST). See *Developing Custom Management Utilities With JMX for Oracle WebLogic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

- [Section 2.4.1, "JDBC MBeans for System Resources"](#)
- [Section 2.4.2, "JDBC Management Objects in the Java EE Management Model \(JSR-77 Support\)"](#)
- [Section 2.4.3, "Using WLST to Create JDBC System Resources"](#)
- [Section 2.4.4, "How to Modify and Monitor JDBC Resources"](#)
- [Section 2.4.5, "Best Practices when Using WLST to Configure JDBC Resources"](#)

2.4.1 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.

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

The WebLogic Server JDBC subsystem supports JSR-77, which defines the Java EE Management Model. The Java EE Management Model is used for monitoring the run-time state of a Java EE Web application server and its resources. You can access the Java EE Management Model to monitor resources, including the WebLogic JDBC 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` and `JDBCDataSourceRuntimeMBeans` currently available in the system.
- `JDBCDriverRuntimeMBean`—Which represents a JDBC driver that the server loaded into memory.
- `JDBCDataSourceRuntimeMBeans`—Which represents a JDBC data source deployed on a server or cluster.

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

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

2.4.3 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"))
.
.
.
```

2.4.4 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.

For more information, see "Navigating MBeans (WLST Online)" in *Oracle WebLogic Scripting Tool*.

2.4.5 Best Practices when Using WLST to Configure JDBC Resources

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

- Trap for Null MBean objects (such as pools, data sources, 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 *Oracle WebLogic Scripting Tool*.

2.5 Creating High-Availability JDBC Resources

You can target or deploy a JDBC resource (a generic data source, multi data source, or GridLink data source) to the members of a cluster using the Administration Console to improve the availability your JDBC resource and load balance communication between resources. However, connections do not fail over in the event that a cluster member becomes unavailable for any reason. New connections are created as needed on available cluster members. See [Section 9, "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).

Configuring JDBC Data Sources

This chapter provides information on how to configure and tune JDBC data sources.

- [Section 3.1, "Understanding JDBC Data Sources"](#)
- [Section 3.2, "Types of WebLogic Server JDBC Data Sources"](#)
- [Section 3.3, "Creating a JDBC Data Source"](#)
- [Section 3.4, "Configuring Connection Pool Features"](#)
- [Section 3.5, "Advanced Connection Properties"](#)
- [Section 3.6, "Configuring Oracle Parameters"](#)
- [Section 3.7, "Configuring an ONS Client"](#)
- [Section 3.8, "Tuning Generic Data Source Connection Pools"](#)
- [Section 3.9, "Setting Database Security Credentials"](#)
- [Section 3.10, "JDBC Data Source Factories \(Deprecated\)"](#)

3.1 Understanding JDBC Data Sources

In WebLogic Server, you configure database connectivity by adding data sources to your WebLogic domain. WebLogic JDBC data sources provide database access and database connection management. Each data source contains a pool of database connections that are created when the data source is created and at server startup. Applications reserve a database connection from the data source by looking up the data source on the JNDI tree or in the local application context and then calling `getConnection()`. When finished with the connection, the application should call `connection.close()` as early as possible, which returns the database connection to the pool for other applications to use.

3.2 Types of WebLogic Server JDBC Data Sources

WebLogic Server provides three types of data sources:

- **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.
- **GridLink Data Sources**—An event-based data source that adaptively responds to state changes in an Oracle RAC instance. See [Section 4, "Using GridLink Data Sources."](#)

- Multi data sources—A *multi data source* is an abstraction around a group of generic data sources that provides load balancing or failover processing. See [Section 5, "Configuring JDBC Multi Data Sources."](#)

3.3 Creating a JDBC Data Source

You can create JDBC data sources in your WebLogic domain using the Administration Console or the WebLogic Scripting Tool (WLST):

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

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

- [Section 3.3.1, "JDBC Data Source Properties"](#)
- [Section 3.3.2, "Configure Transaction Options"](#)
- [Section 3.3.3, "Configure Connection Properties"](#)
- [Section 3.3.4, "Test Connections"](#)
- [Section 3.3.5, "Target the Data Source"](#)

3.3.1 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.

3.3.1.1 Data Source Names

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

3.3.1.2 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. For more information, see "Programming JNDI for Oracle WebLogic Server."

3.3.1.3 Selecting a Database Type

Select a DBMS. For information about supported databases, see "Supported Configurations" in *What's New in Oracle WebLogic Server*.

3.3.1.4 Selecting a JDBC Driver

When creating a JDBC data source using the Administration Console, you are prompted to select a JDBC driver class. The Administration Console provides most of the more common driver class names and in most cases tries to help you construct the URL as required by the driver. 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 Administration Console are shipped (and/or are already in the `classpath`) with WebLogic Server:

- Oracle Thin Driver
 - Oracle Thin Driver XA
 - Oracle Thin Driver non-XA
- Oracle BI Server's Driver (Type 4): Any
- Third-party JDBC drivers (see [Section 12, "Using JDBC Drivers with WebLogic Server"](#)):
 - MySQL (non-XA)
- WebLogic-branded DataDirect drivers for the following database management systems (see *Using WebLogic-branded DataDirect Drivers*):
 - DB2
 - Informix
 - Microsoft SQL Server
 - Sybase

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

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*.

Note: JDBC drivers listed in the Administration 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 Administration Console with known required configuration options to help you configure a data source. The JDBC drivers in the list are not necessarily installed. Driver installation can include setting system Path, Classpath, and other environment variables. See [Section 12.3, "Setting the Environment for a Third-Party JDBC Driver."](#) When a JDBC driver is updated, configuration requirements may change. The Administration 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.

3.3.2 Configure Transaction Options

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

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

Supports Global Transactions: (selected by default) Select this option if you want to use connections from the data source in global transactions, even though you have not selected an XA driver. See [Section 7.1, "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 [Section 7.2, "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 [Section 7.2.4, "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 [Section 7.3, "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.

For more information on configuring transaction support for a data source, see [Section 7, "JDBC Data Source Transaction Options."](#)

3.3.3 Configure Connection Properties

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.

Note: You can use a Single Client Access Name (SCAN) address to represent the host name. When using Oracle RAC 11.2 and higher, consider the following:

- If the Oracle RAC `REMOTE_LISTENER` your data source connects to is set to `SCAN`, the data source connection url can only use a SCAN address.
- If the Oracle RAC `REMOTE_LISTENER` your data source connects to is set to `List of Node VIPs`, the data source connection url can only use a list of VIP addresses.
- If the Oracle RAC `REMOTE_LISTENER` your data source connects to is set to `Mix of SCAN and List of Node VIPs`, the data source connection url can use both SCAN and VIP addresses.

For more information on using SCAN addresses, see "Introduction to Automatic Workload Management" in *Real Application Clusters Administration and Deployment Guide 11g Release 2 (11.2)*.

3.3.3.1 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*.

3.3.4 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.

3.3.5 Target the Data Source

You can select one or more targets 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.

3.4 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: 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 sections include information about connection pool options for a JDBC data source.

- [Section 3.4.1, "Enabling JDBC Driver-Level Features"](#)
- [Section 3.4.2, "Enabling Connection-based System Properties."](#)
- [Section 3.4.3, "Initializing Database Connections with SQL Code"](#)
- [Section 3.5, "Advanced Connection Properties"](#)

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

- **JDBC Data Source: Configuration: Connection Pool** page in the Administration Console. See "JDBC Data Source: Configuration: Connection Pool" in the *Oracle WebLogic Server Administration Console Help*
- `JDBCConnectionPoolParamsBean`, which is a child MBean of the `JDBCDataSourceBean`

3.4.1 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.

3.4.2 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 Administration Console by editing the `System Properties` attribute of your data source configuration.

3.4.3 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 **InitSQL** attribute on the JDBC Data Source: Configuration: Connection Pool page in the Administration 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'
```

or an Informix DBMS:

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

The SQL statement is executed using `JDBC Statement.execute()`. Options that you can set using **InitSQL** vary by DBMS. See the documentation for 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()
```

3.5 Advanced Connection 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 Administration Console by editing the `System Properties` attribute of your data source configuration.

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

- **JDBC Data Source: Configuration: Connection Pool** page in the Administration Console. See "JDBC Data Source: Configuration: Connection Pool" in the *Oracle WebLogic Server Administration Console Help*
- `JDBCConnectionPoolParamsBean`, which is a child MBean of the `JDBCDataSourceBean`

3.5.1 Define Fatal Error Codes

You can define fatal error codes that indicate that the back-end database 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 and the connection is no longer good and 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

To define fatal error codes in the Administration Console, see "Define Fatal Error Codes" in *Oracle WebLogic Server Administration Console Help*.

3.6 Configuring Oracle Parameters

WebLogic Server provides several attributes that provide improved Data Source performance when using Oracle drivers, for more information, see [Section 6, "Advanced Configurations for Oracle Drivers and Databases."](#)

3.7 Configuring an ONS Client

Configuring an ONS client changes a generic data source to a GridLink data source. For more detailed configuration information and additional environment requirements, see [Section 4, "Using GridLink Data Sources."](#)

3.8 Tuning Generic 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. For more information, see [Section 10, "Tuning Data Source Connection Pools."](#)

3.9 Setting Database Security Credentials

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

- [Section 3.9.1, "Types of Data Source Pools"](#)
- [Section 3.9.2, "Using a User Name/Password"](#)
- [Section 3.9.3, "Set Client ID On Connection"](#)

- [Section 3.9.4, "Identity-based Connection Pooling"](#)

3.9.1 Types of Data Source Pools

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

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

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

Table 3–1 Comparing Methods of Passing Security Credentials

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

3.9.2 Using a User Name/Password

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

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

3.9.3 Set Client ID On Connection

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

1. Select `Set Client ID On Connection`, see "Enable Set Client ID On Connection for a JDBC data source" in *Oracle WebLogic Server Administration Console Help*.

Note: Credential mapping to map the WebLogic user ID and the database ID is only supported on the Oracle database with the Oracle Thin driver. This feature is not supported with the Oracle DMS driver.

2. Map the WebLogic user ID and the database ID. See "Configure credential mapping for a JDBC data source" in the *Oracle WebLogic Server Administration Console Help*.

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

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

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

3.9.4 Identity-based Connection Pooling

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

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

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

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

3.9.4.1 How Heterogeneous Connections are Created

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

1. At connection pool initialization, the physical JDBC connections are created with the default DBMS credential of the data source.
2. An application tries to get a connection from a data source.
3. The current server instance credential is mapped to a DBMS credential. See "Configure credential mapping for a JDBC data source" in the *Oracle WebLogic Server Administration Console Help*.
 - If no match is found, the default DBMS credential is used.

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

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

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

3.9.4.2 Using Identity-based Pooling with Global Transactions

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

3.9.4.3 Using Identity-based Pooling with LLR

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

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

3.10 JDBC Data Source Factories (Deprecated)

In previous releases of WebLogic Server, application-scoped JDBC connection pools relied on JDBC data source factories to provide default connection pool values. JDBC data source factories are deprecated in WebLogic Server 9.2 and are included in the release for backward compatibility only. Application-scoped JDBC connection pools are replaced by JDBC application modules. For more information, see [Section A.2.7, "Application Scoping for a Packaged JDBC Module."](#)

Using GridLink Data Sources

This chapter provides information on how to configure and tune GridLink data sources.

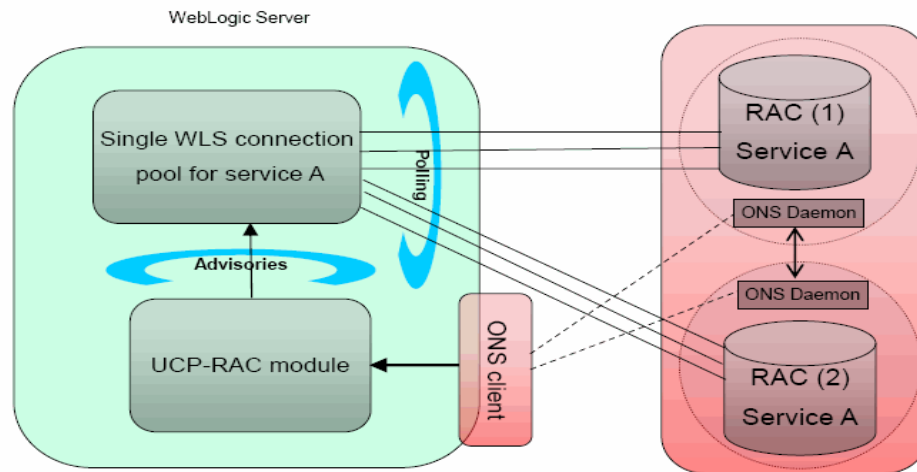
- [Section 4.1, "What is a GridLink Data Source"](#)
- [Section 4.2, "Creating a GridLink Data Source"](#)
- [Section 4.3, "Using Socket Direct Protocol"](#)
- [Section 4.4, "Configuring Connection Pool Features"](#)
- [Section 4.5, "Configuring Oracle Parameters"](#)
- [Section 4.6, "Configuring an ONS Client"](#)
- [Section 4.7, "Configuring Oracle Parameters"](#)
- [Section 4.8, "Tuning GridLink Data Source Connection Pools"](#)
- [Section 4.9, "Setting Database Security Credentials"](#)
- [Section 4.10, "Monitoring GridLink JDBC Resources"](#)
- [Section 4.11, "Best Practices for GridLink Data Sources"](#)

4.1 What is a GridLink Data Source

A single GridLink data source provides connectivity between WebLogic Server and an Oracle Database service, which may include multiple Oracle RAC clusters.¹ It uses the Oracle Notification Service (ONS) to adaptively respond to state changes in an Oracle RAC instance. An Oracle Database service represents a workload with common attributes that enables administrators to manage the workload as a single entity. You scale the number of GridLink data sources as the number of services increases in the data base, independent of the number of nodes in the cluster.

¹ For example, Oracle Data Guard.

Figure 4–1 GridLink Data Source Connectivity



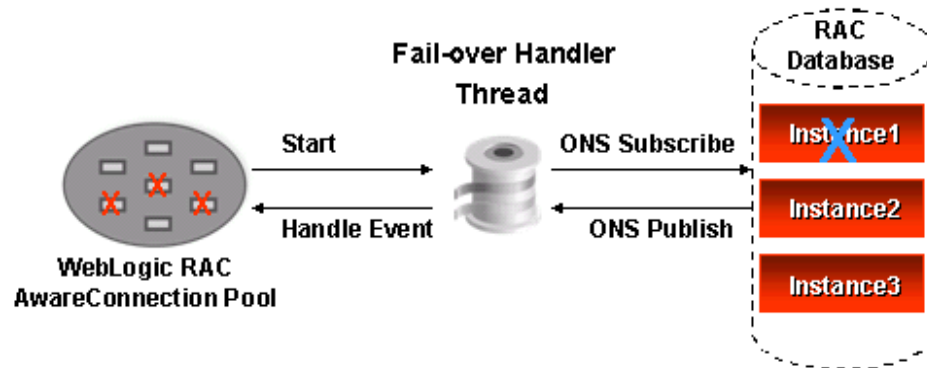
A GridLink data source includes the features of generic data sources plus the following support for Oracle RAC:

- [Section 4.1.1, "Fast Connection Failover"](#)
- [Section 4.1.2, "Runtime Connection Load Balancing"](#)
- [Section 4.1.3, "Graceful Handling for Oracle RAC Outages"](#)
- [Section 4.1.4, "GridLink Affinity"](#)
- [Section 4.1.5, "SCAN Addresses"](#)
- [Section 4.1.6, "Secure Communication using Oracle Wallet."](#)

4.1.1 Fast Connection Failover

A GridLink data source uses Fast Connection Failover and responds to Oracle RAC events using ONS. This ensures that the connection pool in the GridLink data source contains valid connections (including reserved connections) without the need to poll and test connections.

Figure 4–2 Fast Connection Failover



A GridLink 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 [Section 4.1.3, "Graceful Handling for Oracle RAC 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: GridLink 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.

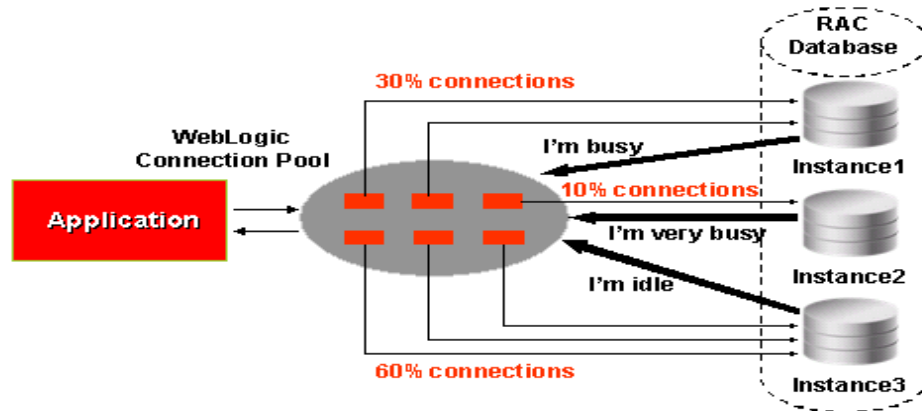
4.1.2 Runtime Connection Load Balancing

GridLink data sources provide load balancing in XA and non-XA environments. GridLink 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 GridLink 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, GridLink data sources use a round-robin load balancing algorithm to allocate connections to Oracle RAC nodes.

Note: Connections may be shut down periodically on 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 of -1 disables the rebalancing process.

4.1.3 Graceful Handling for Oracle RAC Outages

A GridLink data source provides graceful handling for the planned and unplanned shutdown of an Oracle RAC service:

- For planned shutdowns, the data source allows in-progress transactions to complete before closing connections. New Requests are load balanced to active Oracle RAC instances.
- For unplanned shutdowns, the data source rolls back in-progress transactions and closes the connections. New Requests are load balanced to active Oracle RAC instances.

4.1.3.1 Handling for Oracle RAC Outages Prior to Oracle RAC 11.2

In Releases prior to Oracle RAC 11.2, manually shutting down an Oracle RAC instance without first shutting down the corresponding services results in an unplanned shutdown.

4.1.4 GridLink Affinity

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

- [Section 4.1.4.1, "Session Affinity Policy"](#)
- [Section 4.1.4.2, "XA Affinity Policy"](#)

4.1.4.1 Session Affinity Policy

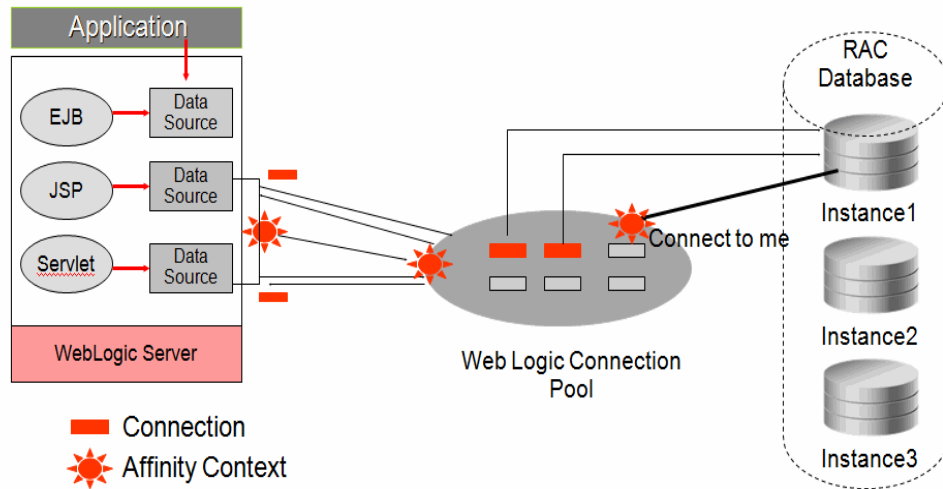
Web applications where a user session has back-to-back online transaction processing (OLTP) have better performance when repeated operations against the same set of records are processed by the same RAC instance. Business applications such as online shopping and online banking are typical examples of this pattern.

A GridLink 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 a GridLink 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 [Section 4.1.4.2, "XA Affinity Policy."](#)

Figure 4–4 Session Affinity



Although the Session Affinity policy for a GridLink data source is always enabled by default, a Web session is active for Session Affinity if:

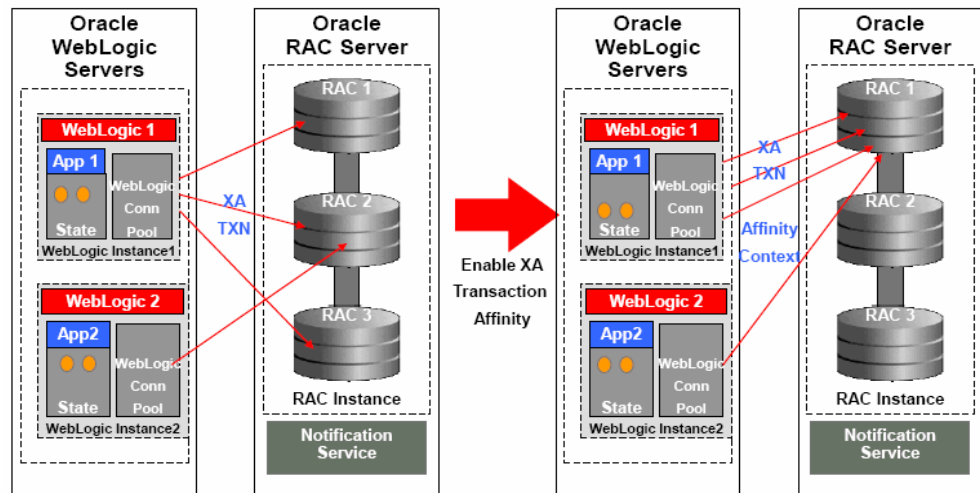
- Oracle RAC is enabled, active, and the service has enabled RCLB. RCLB is enabled for a service if the service GOAL (NOT CLB_GOAL) is set to either SERVICE_TIME or THROUGHPUT.
- The database determines there is sufficient performance improvement in the cluster wait time and the Affinity flag in the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

4.1.4.2 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.

Figure 4–5 XA Affinity



4.1.5 SCAN Addresses

Oracle Single Client Access Name (SCAN) addresses can be used to specify the host and port for both the TNS listener and the ONS listener in the WebLogic console. A GridLink data source containing SCAN addresses does not need to change if you add or remove Oracle RAC nodes. 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 GridLink data source configuration can only use a SCAN address.
- If the Oracle RAC listener is set to `List of Node VIPs`, the GridLink 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 GridLink data source configuration can use both SCAN and VIP addresses.

For more information on using SCAN addresses, see "Introduction to Automatic Workload Management" in Real Application Clusters Administration and Deployment Guide 11g Release 2 (11.2) and <http://www.oracle.com/technetwork/database/clusterin g/overview/scan-129069.pdf>.

4.1.6 Secure Communication using Oracle Wallet

This feature allows you to configure secure communication with the ONS listener using Oracle Wallet. See [Section 4.2.5.1, "Secure ONS Client Communication."](#)

4.2 Creating a GridLink Data Source

To create a GridLink data source in your WebLogic domain, you can use the Administration Console or the WebLogic Scripting Tool (WLST).

See the following for more information:

- "Create a JDBC GridLink Data Source" in the *Oracle WebLogic Server Administration Console Help*.
- The sample WLST script `SAMPLES_HOME\server\examples\src\examples\wlst\online\jdbc_data_source_creation.py`, where `SAMPLES_HOME` refers to the main examples directory of your WebLogic Server installation. This example creates a generic data source. See "WLST Online Sample Scripts" in *Oracle WebLogic Scripting Tool*.

The following sections provide an overview of the basic steps used in the data source configuration wizard to create a data source using the Administration console:

- [Section 4.2.1, "JDBC Data Source Properties"](#)
- [Section 4.2.2, "Configure Transaction Options"](#)
- [Section 4.2.3, "Configure Connection Properties"](#)
- [Section 4.2.4, "Test Connections"](#)
- [Section 4.2.5, "Configure an ONS Client Configuration"](#)
- [Section 4.2.6, "Test ONS Client Configuration"](#)
- [Section 4.2.7, "Target the Data Source"](#)

4.2.1 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.

4.2.1.1 Data Source Names

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

4.2.1.2 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. For more information, see *Programming JNDI for Oracle WebLogic Server*.

4.2.1.3 Select an XA or Non-XA Driver

Specify how your data source handles global transactions.

4.2.2 Configure Transaction Options

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

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

Supports Global Transactions: (selected by default) Select this option if you want to use connections from the data source in global transactions, even though you have not selected an XA driver. See [Section 7.1, "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 [Section 7.2, "Understanding the Logging Last Resource Transaction Option."](#)
- **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 [Section 7.3, "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.

For more information on configuring transaction support for a data source, see [Section 7, "JDBC Data Source Transaction Options."](#)

4.2.3 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.us.myco.com`, the service name `myservice` would need to be entered as `myservice.us.myco.com`.
 - Service names must not include upper case letters.
-
-

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

- [Section 4.2.3.1, "Enter Connection Properties"](#)

- [Section 4.2.3.2, "Enter a Complete URL"](#)

4.2.3.1 Enter Connection Properties

On the **One 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=(ADDRESS=(PROTOCOL=TCP)(HOST=left)(PORT=1234))(ADDRESS=(PRO
TOCOL=TCP)(HOST=right)(PORT=1234))(ADDRESS=(PROTOCOL=TCP)(HOST=c
enter)(PORT=1234)))(CONNECT_DATA=(SERVICE_NAME=myService)))
```

4.2.3.2 Enter a Complete URL

On the **One 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=(ADDRESS=(PROTOCOL=TCP)(HOST=left)(PORT=1234))(ADDRESS=(
PROTOCOL=TCP)(HOST=right)(PORT=1234))(ADDRESS=(PROTOCOL=TCP)(
HOST=center)(PORT=1234)))(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.co
m)(PORT=1234)))(CONNECT_DATA=(SERVICE_NAME=myService)))
```

- Enter **myDataBase** in `Database User Name`.
- Enter **myPassword1** in `Password`.
- If required, set **Protocol** to `SDP`.

4.2.4 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.

4.2.5 Configure an ONS Client Configuration

ONS client configuration allows the data source to subscribe to and process Oracle FAN events. To configure an ONS client:

- Select `Fan Enabled`.
- In **ONS host and port**, enter a comma-separated list of ONS daemon listen addresses and ports for receiving ONS-based FAN events. You can use Single Client Access Name (SCAN) addresses to access FAN notifications.
- Optionally, configure secure ONS client communication using SSL. See [Section 4.2.5.1, "Secure ONS Client Communication."](#)

4.2.5.1 Secure ONS Client Communication

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

- Add the following files to the WebLogic Server classpath:
 - `$MW_HOME/modules/com.oracle.osdt_cert_1.0.0.0.jar`
 - `$MW_HOME/modules/com.oracle.osdt_core_1.0.0.0.jar`
 - `$MW_HOME/modules/com.oracle.oraclepki_1.0.0.0.jar`
- Update your GridLink Data Source configuration to include the directory of the Oracle wallet file in which the SSL certificates are stored and the ONS Wallet password. See *Secure ONS Listener using Oracle Wallet* in *Oracle WebLogic Server Administration Console Help*.
- For more information on Oracle Wallet, see the *Database Advanced Security Administrator's Guide*.

4.2.6 Test ONS Client Configuration

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

4.2.7 Target the Data Source

You can select one or more targets to deploy your new JDBC 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.

4.3 Using Socket Direct Protocol

To use 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*.

4.3.1 Configuring Runtime Load Balancing

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.us.myco.com) (PORT=1522))
      (ADDRESS = (PROTOCOL = SDP) (HOST =
        sclcgdb02-bvip.us.myco.com) (PORT=1522))
    )
  )
```

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

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=SDP)
(HOST=sclcgdb01-bvip.us.myco.com) (PORT=1522)) (ADDRESS=(PROTOCOL=SDP)
(HOST=sclcgdb02-ibvip.us.myco.com) (PORT=1522))) (CONNECT_DATA=(SERVICE_
NAME=elservice)))
```

4.4 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: 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 sections include information about connection pool options for a JDBC data source.

- [Section 4.4.1, "Enabling JDBC Driver-Level Features"](#)
- [Section 4.4.2, "Enabling Connection-based System Properties."](#)
- [Section 4.4.3, "Initializing Database Connections with SQL Code"](#)

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

- **JDBC Data Source: Configuration: Connection Pool** page in the Administration Console. See "JDBC Data Source: Configuration: Connection Pool" in the *Oracle WebLogic Server Administration Console Help*
- `JDBCConnectionPoolParamsBean`, which is a child MBean of the `JDBCDataSourceBean`

4.4.1 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.

4.4.2 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 Administration 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.

4.4.3 Initializing Database Connections with SQL Code

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

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

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

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

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

or

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

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

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

4.5 Configuring Oracle Parameters

WebLogic Server provides several attributes that provide improved Data Source performance when using Oracle drivers, for more information, see [Section 6, "Advanced Configurations for Oracle Drivers and Databases."](#)

4.6 Configuring an ONS Client

The following section provides information on how to configure an ONS client.

- [Section 4.6.1, "Enabling FAN Events"](#)
- [Section 4.6.2, "Configuring Generic Data Source Connection Testing."](#)
- [Section 4.6.3, "Using a Wallet File"](#)

4.6.1 Enabling FAN Events

Enabling a data source to subscribe to and process Oracle Fast Application Notification (FAN) events.

1. Select `Fan Enabled`
2. Provide a comma-separated list of ONS daemon listen addresses and ports for receiving ONS-based FAN events. You can use Single Client Access Name (SCAN) addresses to access FAN notifications.

See Configure ONS client parameters in *Oracle WebLogic Server Administration Console Help*.

4.6.2 Configuring Generic Data Source Connection Testing

Normally, when FAN events are enabled, it's not necessary to enable test on reserve. However, if the ONS server is down or in unhealthy state, you can disable `FAN Enabled` and enable test on reserve. For more information, see [Section 10.2, "Connection Testing Options for a Data Source."](#)

4.6.3 Using a Wallet File

To communicate with ONS daemons using SSL, you must use a wallet file. See [Section 4.2.5.1, "Secure ONS Client Communication."](#)

4.7 Configuring Oracle Parameters

WebLogic Server provides several attributes that provide improved Data Source performance when using Oracle drivers, for more information, see [Section 6, "Advanced Configurations for Oracle Drivers and Databases."](#)

4.8 Tuning 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. For more information, see [Section 10, "Tuning Data Source Connection Pools."](#)

4.9 Setting Database Security Credentials

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

- [Section 4.9.1, "Using a User Name/Password"](#)
- [Section 4.9.2, "Set Client ID On Connection"](#)

Note: GridLink Data Sources do not support identity-based connection pooling.

4.9.1 Using a User Name/Password

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

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

4.9.2 Set Client ID On Connection

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

1. Select `Set Client ID On Connection`, see "Enable Set Client ID On Connection for a JDBC data source" in *Oracle WebLogic Server Administration Console Help*.

Note: Credential mapping to map the WebLogic user ID and the database ID is only supported on the Oracle database with the Oracle Thin driver. This feature is not supported with the Oracle DMS driver.

2. Map the WebLogic user ID and the database ID. See "Configure credential mapping for a JDBC data source" in the *Oracle WebLogic Server Administration Console Help*.

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

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

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

4.10 Monitoring GridLink JDBC Resources

The following sections include details about monitoring GridLink JDBC objects:

- [Section 4.10.1, "Viewing Run-Time Statistics"](#)
- [Section 4.10.2, "Debug GridLink Data Sources."](#)

For more information on JDBC monitoring, see [Section 13, "Monitoring WebLogic JDBC Resources."](#)

4.10.1 Viewing Run-Time Statistics

You can view run-time statistics for a GridLink data source via the Administration Console or through the associated runtime MBeans.

4.10.1.1 JDBCOracleDataSourceInstanceRuntimeMBean

The `JDBCOracleDataSourceInstanceRuntimeMBean` provides methods for getting the current state of the data source instance. For more information, see "`JDBCOracleDataSourceInstanceRuntimeMBean`" in the *Oracle WebLogic Server MBean Reference*.

4.10.1.2 JDBCDataSourceRuntimeMBean

The `JDBCDataSourceRuntimeMBean` provides methods for getting the current state of the data source instance. 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, and the highest number of active connections. For more information, see "`JDBCDataSourceRuntimeMBean`" in the *Oracle WebLogic Server MBean Reference*.

4.10.1.3 ONSDaemonRuntimeMBean

The `ONSDaemonRuntimeMBean` provides methods for monitoring the ONS client configuration that is associated with a GridLink data source. For more information, see "`ONSDaemonRuntimeMBean`" in the *Oracle WebLogic Server MBean Reference*.

4.10.2 Debug GridLink Data Sources

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

4.10.2.1 JDBC Debugging Scopes

The following are registered debugging scopes for JDBC:

- `DebugJDBCRCAC` (scope `weblogic.jdbc.racl`) - prints information about GridLink data source lifecycle, UCP callback, and connection information.
- `DebugJDBCONS` (scope `weblogic.jdbc.connection`) - traces ONS client information, including the LBA event body.

4.10.2.2 UCP JDK Logging

You can enable UPC JDK logging by following the instructions at http://download.oracle.com/docs/cd/B28359_01/java.111/e10788/get_started.htm#sthref67.

4.10.2.3 Enable Debugging Using the Command Line

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

```
-Dweblogic.debug.DebugJDBCRCAC=true  
-Dweblogic.debug.DebugJDBCRCONS=true  
-Dweblogic.debug.DebugJDBCUCP=true
```

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

4.11 Best Practices for GridLink Data Sources

The following sections provide best practices for using GridLink data sources:

- [Section 4.11.1, "Catch and Handle Exceptions"](#)

4.11.1 Catch and Handle Exceptions

Applications need to catch and handle all exceptions. Applications using 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 pool. For unplanned down events, a connection pool aborts all borrowed connections that are affected by the outage.

Configuring JDBC Multi Data Sources

This chapter provides information on how to configure and use a multi data source to provides load balancing or failover processing at the time of connection requests, between the data sources associated with the multi data source.

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

This section includes the following information:

- [Section 5.1, "Multi Data Source Features"](#)
- [Section 5.2, "Creating and Configuring Multi Data Sources"](#)
- [Section 5.3, "Choosing the Multi Data Source Algorithm"](#)
- [Section 5.4, "Multi Data Source Fail-Over Limitations and Requirements"](#)
- [Section 5.5, "Multi Data Source Failover Enhancements"](#)
- [Section 5.6, "Deploying JDBC Multi Data Sources on Servers and Clusters"](#)

5.1 Multi Data Source Features

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

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

- Grow and shrink Oracle RAC clusters in response to throughput. See [Section 5.1.2, "Adding a Database Node."](#)
- Shutdown Oracle RAC nodes for maintenance. See [Section 5.1.1, "Removing a Database Node."](#)

See [Section C, "Using Multi Data Sources with Oracle RAC."](#)

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.

5.1.1 Removing a Database Node

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

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

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

5.1.2 Adding a Database Node

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

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

5.2 Creating and Configuring Multi Data Sources

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

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

For information about the configuration files created when configuring a multi data source, see [Section 2.1, "Understanding JDBC Resources in WebLogic Server."](#) Also see [Section A.2.5, "Creating a JDBC Multi Data Source Module."](#)

5.3 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.

5.3.1 Failover

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

Note: This algorithm requires that Test Reserved Connections (`TestConnectionsOnReserve`) on the data source is enabled. If enabled, a connection in the first data source is tested to verify if the data source is healthy. If the connection fails the test, the multi data source uses a connection from the next data source listed in the multi data source. See [Section 10.2, "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.

5.3.2 Load Balancing

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

5.4 Multi Data Source Fail-Over Limitations and Requirements

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

5.4.1 Test Connections on Reserve to Enable Fail-Over

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

5.4.2 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.

5.5 Multi Data Source Failover Enhancements

The following enhancements improve failover processing for multi data sources:

- Connection request routing enhancements to avoid requesting a connection from an automatically disabled (dead) data source within a multi data source. See [Section 5.5.1, "Connection Request Routing Enhancements When a Data Source Fails."](#)
- Automatic failback on recovery of a failed data source within a multi data source. See [Section 5.5.2, "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source."](#)
- Failover for busy data sources within a multi data sources. See [Section 5.5.3, "Enabling Failover for Busy Data Sources in a Multi Data Source."](#)
- Failover callbacks for multi data sources with the Failover algorithm. See [Section 5.5.4, "Controlling Multi Data Source Failover with a Callback."](#)
- Failback callbacks for multi data sources with either algorithm. See [Section 5.5.5, "Controlling Multi Data Source Failback with a Callback."](#)

5.5.1 Connection Request Routing Enhancements When a Data Source Fails

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

This feature requires that data source testing options are configured for all data sources in a multi data source, specifically Test Table Name and Test Reserved Connections. See [Section 10.2, "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 data source in the list. See [Section 5.5.4, "Controlling Multi Data Source Failover with a Callback"](#) for more details.

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

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

will test disabled data sources every 120 seconds. See "JDBC Multi Data Source: Configuration: General" in the *Oracle WebLogic Server Administration Console Help*.

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

If a callback handler is registered for the multi data source, WebLogic Server calls the callback handler before re-enabling the data source. See [Section 5.5.5, "Controlling Multi Data Source Failback with a Callback"](#) for more details.

5.5.3 Enabling Failover for Busy Data Sources in a Multi Data Source

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

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

If a `ConnectionPoolFailoverCallbackHandler` is included in the multi data source configuration, WebLogic Server calls the callback handler before failing over. See [Section 5.5.4, "Controlling Multi Data Source Failover with a Callback"](#) for more details.

5.5.4 Controlling Multi Data Source Failover with a Callback

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

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

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

5.5.4.1 Callback Handler Requirements

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

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

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

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

5.5.4.2 Callback Handler Configuration

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

- **Failover Callback Handler** (`ConnectionPoolFailoverCallbackHandler`)—To register a failover callback handler for a multi data source, you add a value for this attribute to the multi data source configuration. The value must be an absolute name, such as `com.bea.samples.wls.jdbc.MultiDataSourceFailoverCallbackApplication`. You can set the Failover Callback Handler using the Administration Console (see "Register a failover callback handler" in the *Oracle WebLogic Server Administration Console 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) data sources to see if they are now available. See [Section 5.5.2, "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source"](#) for more details.

5.5.4.3 How It Works—Failover

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

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

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

- `currPool`—For failover, this is the name of data source currently being used to supply database connections. This is the "failover from" data source.
- `nextPool`—The name of next available data source listed in the multi data source. For failover, this is the "failover to" data source.
- `opcode`—A code that indicates the reason for the call:
 - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current data source is dead and has disabled it.
 - `OPCODE_CURR_POOL_BUSY`—All database connections in the data source are in use. (Requires `FailoverIfBusy=true` in the multi data source configuration. See [Section 5.5.3, "Enabling Failover for Busy 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 data source in the list.
- `RETRY_CURRENT`—Retry the connection request with the current data source.
- `DONOT_FAILOVER`—Do not retry the current connection request and do not failover. WebLogic Server will throw a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

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

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

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

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

5.5.5 Controlling Multi Data Source Failback with a Callback

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

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

- [Section 5.5.4.1, "Callback Handler Requirements"](#)
- [Section 5.5.4.2, "Callback Handler Configuration"](#)

5.5.5.1 How It Works—Failback

WebLogic Server periodically checks the status of data sources in a multi data source that were automatically disabled. (See [Section 5.5.2, "Automatic Re-enablement on Recovery of a Failed Data Source within a Multi Data Source."](#)) If a disabled data source becomes available and if a failover callback handler is registered, WebLogic Server calls the callback handler with the following information and waits for a return:

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

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

The callback handler can return one of the following values:

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

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

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

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

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

5.6 Deploying JDBC Multi Data Sources on Servers and Clusters

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

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

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

Advanced Configurations for Oracle Drivers and Databases

This chapter provides advanced configuration options that can provide improved data source and driver performance when using Oracle drivers and databases. Options include proxy authentication, setting credentials on a connection, connection harvesting, and labeling connections.

- [Section 6.1, "Options to Improve Driver Performance"](#)
- [Section 6.2, "Proxy Authentication for Oracle Databases"](#)
- [Section 6.3, "Setting Database Credentials on a Connection"](#)
- [Section 6.4, "Configuring Connection Harvesting"](#)
- [Section 6.5, "Labeling Connections"](#)

6.1 Options to Improve Driver Performance

WebLogic Server provides several attributes that provide improved Data Source performance when using Oracle drivers, including:

- **Oracle Enable JavaNet Fastpath:** Enables the Oracle JDBC JavaNet Fastpath to reduce data copies and fragmentation.
- **Oracle Optimize UTF8 Conversion:** Enables the Oracle JDBC optimize UTF-8 conversion option.

If your environment supports these attributes, see Configure Oracle parameters in the *Oracle WebLogic Server Administration Console Help*.

6.2 Proxy Authentication for Oracle Databases

Proxy authentication allows one JDBC connection to act as a proxy for other JDBC connections to an Oracle database. 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.

6.2.1 How to Configure Proxy Authentication

Use the following steps to configure proxy authentication to an Oracle database:

1. If you have not yet done so, create the necessary WebLogic Server and database users.

2. On the Oracle database, provide `CONNECT THROUGH` privileges.

For example:

```
SQL> ALTER USER wlsuser GRANT CONNECT THROUGH dbuser;
```

where *wlsuser* is a WebLogic Server user and *dbuser* is an Oracle database user.

3. On a WebLogic Server instance:
 1. Create a generic or GridLink data source and set the user to the value of *dbuser*.
 2. Map the value of *wlsuser* to the value of *dbuser*. See "Configure credential mapping for a JDBC data source" in the *Oracle WebLogic Server Administration Console Help*.
 3. Redeploy the data source.
4. Log on to a WebLogic Server instance using the value of *wlsuser*.
5. Enable **Proxy Authentication**, see "Configure Oracle parameters" in *Oracle WebLogic Server Administration Console Help*.

A proxy session is opened on the connection for the application-authenticated user each time a connection request is made on the pool. The proxy session is closed when the connection is returned to the pool.

Note: Opening or closing a proxy session:

- Closes any existing statements (including result sets) from the original connection.
- Clears the WebLogic Server statement cache.
- 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.

6.3 Setting Database Credentials on a Connection

You can configure a WebLogic data source to allow a connection to set database credentials when connecting to an Oracle database. When configured, a proxy session is opened for a connection created using `getConnection(username, password)` to specify a database user and password. This provides a simple, light-weight mechanism to authenticate many users and avoids the need to credential mapping entries for each database user.

Use the following steps to configure proxy authentication of database credentials set 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. On a WebLogic Server instance, create a generic or GridLink data source and set the user to the value of *dbuser*.
4. Enable **Use Database Credentials**, see "Configure Oracle parameters" in *Oracle WebLogic Server Administration Console Help*.

A proxy session is opened on the connection for the application-authenticated user each time a connection request is made on the pool. The proxy session is closed when the connection is returned to the pool.

Note: Opening or closing a proxy session:

- Closes any existing statements (including result sets) from the original connection.
- Clears the WebLogic Server statement cache.
- 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.

5. Open a connection using `getConnection(username, password)`.

Note: `getConnection` fails if **Use Database Credentials** is not enabled and the value of *connectionuser* is not a WebLogic Server user.

6.3.1 Considerations When Setting Database Credentials on a Connection

Consider the following when setting database credentials on a connection that participates in transactions:

- When using `getConnection(username, password)` with data sources configured with non-XA LLR or 1PC (JTS driver) with global transactions, the first connection obtained within the transaction is returned on subsequent `getConnection()` requests regardless of the values of *username/password* specified. The connection must use the proxy user session information from the first `getConnection()` request because the connection must be shared among all users of the connection when using LLR or 1PC.
- For XA data sources, connections may be stored with the transaction context on a given JVM and subsequent `getConnection()` requests may return a connection with a proxy user for a prior `getConnection()` request. Oracle recommends that a single database *username* be specified when using this feature in global transactions.

6.4 Configuring Connection Harvesting

You can specify a number of reserved connections to be released when a data source reaches a specified number of available connections. Harvesting helps to ensure that a specified number of connections are always available in the pool and improves performance by minimizing connection initialization.

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.

Use the following steps to use connection harvesting in your applications:

1. [Enable Connection Harvesting](#)
2. [Make Connections Harvestable](#)
3. [Recover Harvested Connections](#)

6.4.1 Enable Connection Harvesting

The `Connection Harvest Trigger Count` attribute of a data source configuration is used to enable and specify a threshold to trigger connection harvesting. 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 indicates that connection harvesting is disabled.

When connection harvesting is triggered, the `Connection Harvest Max Count` specifies how many reserved connections can be returned to the pool. The number of connections actually harvested ranges from 1 to the value of `Connection Harvest Max Count`.

See "Configure connection harvesting for a connection pool" in *Oracle WebLogic Server Administration Console Help*.

6.4.2 Make Connections Harvestable

When connection harvesting is enabled, all connections are 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 the transaction completes, 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()`.

6.4.3 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.
 - Harvesting is done as part of the data source maintenance processing. The data source maintenance interval is fixed by default to the value of `SecurityCacheTimeoutSeconds`.
-

6.5 Labeling Connections

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. 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. 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.
- Configuring the data source to keep connections after transactions, see "JDBC Data Source: Configuration: Transaction" in *Oracle WebLogic Server Administration Console Help*.

6.5.1 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 *Programming JDBC for Oracle WebLogic Server*.

6.5.2 Creating a Labeling Callback

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 `getRequestLabels()` method. `getRequestLabels` 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.

6.5.2.1 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 6–1 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;
}
}
```

6.5.3 Registering a Labeling Callback

A WLS data source provides the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. Only one callback may be registered on a connection pool. 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 Administration Console, see "Configure a connection labeling callback class" in *Oracle WebLogic Server Administration Console Help*.

6.5.3.1 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. See "Configure a connection labeling callback class" in *Oracle WebLogic Server Administration Console Help*

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.

6.5.3.2 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 [Section 6.5.2, "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);
. . .
```

6.5.4 Reserving Labeled Connections

A WLS data source provides two `getConnection` methods that are used to reserve a labeled connection from the pool. The methods are shown below:

```
public Connection getConnection(java.util.Properties labels)
    throws SQLException;

public Connection getConnection(String user, String password,
    java.util.Properties labels) throws SQLException;
```

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.

6.5.5 Checking Unmatched labels

A connection may have multiple labels that each uniquely identifies the connection based on some desired criteria. The `getUnmatchedConnectionLabels` method is used to verify which connection labels matched from the requested labels and which did not. The method is used after a connection with multiple labels is reserved from the connection pool and is typically used by a labeling callback. The following code example demonstrates checking for unmatched labels:

```
. . .
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
```



```
label.setProperty(pname, pvalue);
. . .
Connection conn = ((WLDataSource)ds).getConnection(label);
Properties unmatched =
    ((LabelableConnection)connection).getUnmatchedConnectionLabels (label);
. . .
```

6.5.6 Removing a Connection Label

The `removeConnectionLabel` method is used to remove a label from a connection. This method is used after a labeled connection is reserved from the connection pool. 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);
. . .
```

JDBC Data Source Transaction Options

This chapter provides information on XA, non-XA, and Global Transaction options for WebLogic data sources.

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

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

Supports Global Transactions: (selected by default) Select this option if you want to use connections from the data source in global transactions, even though you have not selected an XA driver. See [Section 7.1, "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 [Section 7.2, "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 [Section 7.2.4, "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 [Section 7.3, "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.

This section includes the following information:

- [Section 7.1, "Enabling Support for Global Transactions with a Non-XA JDBC Driver"](#)
- [Section 7.2, "Understanding the Logging Last Resource Transaction Option"](#)
- [Section 7.3, "Understanding the Emulate Two-Phase Commit Transaction Option"](#)

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

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

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

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

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

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

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

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

7.2 Understanding the Logging Last Resource Transaction Option

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

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

The LLR optimization improves performance by:

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

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

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

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

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

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

7.2.1 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 *Programming JTA for Oracle WebLogic Server*.

7.2.2 Enabling the Logging Last Resource Transaction Optimization

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

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

7.2.3 Programming Considerations and Limitations for LLR Data Sources

You use JDBC connections from an LLR-enabled data source in an application as you would use JDBC connections from any other data source: *after* beginning a transaction, you look up the data source on the JNDI tree, then request a connection from the data source. However, with the LLR optimization, WebLogic Server internally manages the connection request and handles the transaction processing differently than in an XA transaction. For more information about how Logging Last Resource works, see "Logging Last Resource Transaction Optimization" in *Programming JTA 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 *Programming JTA 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 [Section 7.3.1.3, "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. (See "Oracle WebLogic Extensions to JTA" in *Programming JTA for Oracle WebLogic Server*) Your database administrator can manually create a table with larger columns if this occurs.

7.2.4 Administrative Considerations and Limitations for LLR Data Sources

Consider the following requirements and limitations when configuring an LLR-enabled JDBC data source. For more information about how Logging Last Resource works, see "Logging Last Resource Transaction Optimization" in *Programming JTA 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_LLRSERVERNAME`. You can configure the table name in the Administration Console on the Server > Configuration > General tab under Advanced options. See "Servers: Configuration: General" in *Oracle WebLogic Server Administration Console Help*.
- 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 *Programming JTA 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. LLR is not supported with any other multi data source configuration.

- If you use credential mapping on an LLR data source, all mapped users must have write permissions on the LLR table.
- You cannot use a JDBC XA driver to create database connections in a JDBC LLR data source. If the JDBC driver used in a JDBC LLR data source supports XA, a warning message is logged, and the data source participates in transactions as a full XA resource rather than as an LLR resource.
- Transaction statistics for LLR resources are tracked under "NonXAResource." See "View transaction statistics for non-XA resources" in the *Oracle WebLogic Server Administration Console Help*.
- 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.

7.3 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 *Programming JDBC for Oracle WebLogic Server*.

7.3.1 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.

7.3.1.1 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.

7.3.1.2 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.

7.3.1.3 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.

7.3.1.4 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.

Using Roles and Policies to Secure JDBC Data Sources

This chapter provides information on how WebLogic Server uses roles and policies to secure JDBC data sources.

- [Section 8.1, "Setting Security Policies for JDBC Resources"](#)
- [Section 8.2, "Security Roles for JDBC MBeans"](#)

8.1 Setting Security Policies for JDBC Resources

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

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. See "Java Data Base Connectivity (JDBC) Resources" in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

8.2 Security Roles for JDBC MBeans

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

- [Section 8.2.1, "JDBC Domain Configuration MBeans"](#)
- [Section 8.2.2, "JDBC System Module MBeans"](#)

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

8.2.1 JDBC Domain Configuration MBeans

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

- JDBCConnectionPoolMBean (deprecated)
- JDBCDataSourceFactoryMBean (deprecated)

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

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

8.2.2 JDBC System Module MBeans

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

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

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

Deploying Data Sources on Servers and Clusters

This chapter provides information on how to deploy data sources on servers and clusters.

- [Section 9.1, "Deploying Data Sources on Servers and Clusters"](#)
- [Section 9.2, "Minimizing Server Startup Hang Caused By an Unresponsive Database"](#)

9.1 Deploying Data Sources on Servers and Clusters

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

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

9.2 Minimizing Server Startup Hang Caused By an Unresponsive Database

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

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

Tuning Data Source Connection Pools

This chapter provides information on how to properly tune the connection pool attributes in JDBC data sources in your WebLogic Server domain to improve application and system performance.

The following sections include information about tuning options for the connection pool in a JDBC data source:

- [Section 10.1, "Increasing Performance with the Statement Cache"](#)
- [Section 10.2, "Connection Testing Options for a Data Source"](#)
- [Section 10.3, "Enabling Connection Creation Retries"](#)
- [Section 10.4, "Enabling Connection Requests to Wait for a Connection"](#)
- [Section 10.5, "Automatically Recovering Leaked Connections"](#)
- [Section 10.6, "Avoiding Server Lockup with the Correct Number of Connections"](#)
- [Section 10.7, "Limiting Statement Processing Time with Statement Timeout"](#)
- [Section 10.8, "Using Pinned-To-Thread Property to Increase Performance"](#)
- [Section 10.9, "Using Unwrapped Data Type Objects"](#)

10.1 Increasing Performance with the Statement Cache

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

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

- **Statement Cache Type**—The algorithm that determines which statements to store in the statement cache. See [Section 10.1.1, "Statement Cache Algorithms."](#)
- **Statement Cache Size**—The number of statements to store in the cache for each connection. The default value is 10. See [Section 10.1.2, "Statement Cache Size."](#)

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

10.1.1 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:

- [Section 10.1.1.1, "LRU \(Least Recently Used\)"](#)
- [Section 10.1.1.2, "Fixed"](#)

10.1.1.1 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.

10.1.1.2 Fixed

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

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

10.1.2 Statement Cache Size

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

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

10.1.3 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.

10.1.3.1 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.

10.1.3.2 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)
```

10.1.3.3 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.

10.2 Connection Testing Options for a Data Source

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

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

The following section discusses automatic connection testing options. For more information about manual connection testing, see [Section 14.1, "Testing Data Sources and Database Connections."](#)

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

- **Test Frequency**—(`TestFrequencySeconds` in the `JDBCConnectionPoolParamsBean`) Use this attribute to specify the number of seconds between tests of unused connections. WebLogic Server tests unused 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.
- **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 [Section 10.2.1.7, "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection."](#)

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

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

10.2.1 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 connect, 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.

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). If a pool disables itself in this manner, it will periodically try to reconnect to the DBMS (every 5 seconds by default). Once a new connection can be made, the pool will re-enable itself, and subsequent connection requests will be served as normal (with the pool repopulating itself as load requires).

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

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

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

- [Section 10.2.1.1, "Connection Testing When Database Connections are Created"](#)
- [Section 10.2.1.2, "Periodic Connection Testing"](#)
- [Section 10.2.1.3, "Testing Reserved Connections"](#)
- [Section 10.2.1.4, "Minimized Connection Test Delay After Database Connectivity Loss"](#)
- [Section 10.2.1.5, "Minimized Connection Request Delay After Connection Test Failures"](#)
- [Section 10.2.1.6, "Minimized Connection Request Delays After Loss of DBMS Connectivity"](#)
- [Section 10.2.1.7, "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection"](#)

10.2.1.1 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.

10.2.1.2 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.

10.2.1.3 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 [Section 10.2.1.7, "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection."](#)

10.2.1.4 Minimized Connection Test Delay After Database Connectivity Loss

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

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

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

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

10.2.1.5 Minimized Connection Request Delay After Connection Test Failures

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

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

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

- The server instance executes a health check on the database server every 5 seconds. This setting is not configurable.

- If the server instance recognizes that the database was recovered, it creates a new database connection and enables the data source.

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

10.2.1.6 Minimized 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, testing a dead database connection and trying futilely to replace it can in some cases take as long as the OS network timeout (minutes), and can cause long delays to clients before getting the expected failure message.

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.

10.2.1.7 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 "Configure testing options for a JDBC data source" in the *Oracle WebLogic Server Administration Console 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.

10.2.2 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.

10.2.3 Default Test Table Name

When you create a data source using the Administration Console, the Administration Console automatically sets the Test Table Name attribute for a data source based on the DBMS that you select. The Test Table Name attribute is used in connection testing which is optionally performed periodically or when you create or reserve a connection, depending on how you configure the testing options. For database tests to succeed, the database user used to create database connections in the data source must have access to the database table. If not, you should either grant access to the user (make this change in the DBMS) or change the Test Table Name attribute to the name of a table to which the user does have access (make this change in the WebLogic Server Administration Console).

The Test Table Name is an overloaded parameter. Its simplest form is to name a table that WLS will query to test a connection. Setting it to any table, such as "DUAL" for Oracle, will cause 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 WLS pool connection testing, and to make the test as fast as possible.

Table 10–1 Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
Adabas for z/OS	SQL call shadow_adabas('select * from employees')
Cloudscape	SQL SELECT 1
DB2	SQL SELECT COUNT(*) FROM SYSIBM.SYSTABLES
FirstSQL	SQL SELECT 1
IMS/TM for z/OS	SQL call shadow_ims('otm','/dis','ctl')
Informix	SQL SELECT COUNT(*) FROM SYSTABLES

Table 10–1 (Cont.) Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
Microsoft SQL Server	SQL SELECT 1
MySQL	SQL SELECT 1
Oracle	SQL SELECT 1 FROM DUAL
PostgreSQL	SQL SELECT 1
Progress	SQL SELECT COUNT(*) FROM SYSTABLES
Sybase	SQL SELECT 1

10.3 Enabling Connection Creation Retries

WebLogic JDBC data sources include the Connection Creation Retry Frequency option (`ConnectionCreationRetryFrequencySeconds` in the `JDBCConnectionPoolParamsBean`) that you can use to specify the number of seconds between attempts to establish connections to the database. If set and if the database is unavailable when the data source is created, WebLogic Server attempts to create connections in the pool again after the number of seconds you specify, and will continue to attempt to create the connections until it succeeds. This option applies to connections created when the data source is created at server startup or when the data source is deployed or if the initial capacity is increased. It does *not* apply to connections created for pool expansion or to replace a defunct connection in the pool.

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

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

10.4 Enabling Connection Requests to Wait for a Connection

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

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

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

10.4.1 Connection Reserve Timeout

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

application will get a `PoolLimitSQLException` exception.

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

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

10.4.2 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.

10.5 Automatically Recovering Leaked Connections

A leaked connection is a connection that was not properly returned to the connection pool in the data source. To automatically recover leaked connections, you can specify a value for `Inactive Connection Timeout` on the JDBC Data Source:

Configuration: `Connection Pool` page in the Administration Console. When you set a value for `Inactive Connection Timeout`, WebLogic Server 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.

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

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

10.6 Avoiding Server Lockup with the Correct Number of Connections

When your applications attempt to get a connection from a data source in which there are no available connections, the data source throws an exception stating that a connection is not available in the data source. To avoid this error, make sure your data source can expand to the size required to accommodate your peak load of connection requests. To increase the maximum number of connections available in the data source, increase the value for `Maximum Capacity` for the data source on the `JDBC Data Source: Configuration: Connection Pool` page in the Administration Console.

10.7 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.

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

10.8 Using Pinned-To-Thread Property to Increase Performance

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

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

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

See "JDBC Data Source: Configuration: Connection Pool" in the *Oracle WebLogic Server Administration Console Help*.

10.8.1 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.

10.8.2 Additional Database Resource Costs When `PinnedToThread` is Enabled

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

Also note that connections are never returned to the connection pool, which means that the connection pool can never shrink to reduce the number of connections and associated resources in use. You can minimize this cost by setting an additional driver parameter `onePinnedConnectionOnly`. When `onePinnedConnectionOnly=true`, only the first connection requested is pinned to the thread. Any additional connections required by the thread are taken from and returned to the connection pool as needed. Set `onePinnedConnectionOnly` using the `Properties` attribute, for example:

```
Properties="PinnedToThread=true;onePinnedConnectionOnly=true;user=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`.

10.9 Using Unwrapped Data Type Objects

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 wrapping 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.
- 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.

10.9.1 How to Disable Wrapping

You can use the Administration Console and WLST to disable data type wrapping.

10.9.1.1 Disable Wrapping using the Administration Console

To disable wrapping of JDBC data type objects:

1. If you have not already done so, in the Change Center of the Administration Console, click **Lock & Edit**.
2. In the **Domain Structure** tree, expand **Services**, then select **Data Sources**.
3. On the Summary of Data Sources page, click the data source name.
4. Select the **Configuration: Connection Pool** tab.
5. Scroll down and click **Advanced** to show the advanced connection pool options.
6. In **Wrap Data Types**, deselect the checkbox to disable wrapping.
7. Click Save.
8. To activate these changes, in the Change Center of the Administration Console, click **Activate Changes**.

This change does not take effect immediately—it requires that the data source be redeployed or the server be restarted.

10.9.1.2 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.

Using WebLogic Server with Oracle RAC

This chapter describes the requirements and configuration tasks for using Oracle Real Application Clusters (Oracle RAC) with WebLogic Server:

Oracle WebLogic Server provides strong support for Oracle Real Application Clusters (RAC) features in Oracle Database 11g, minimizing database access time while allowing transparent access to rich pooling management functions that maximizes both connection performance and availability.

- [Section 11.1, "Overview of Oracle Real Application Clusters"](#)
- [Section 11.2, "Software Requirements"](#)
- [Section 11.3, "JDBC Driver Requirements"](#)
- [Section 11.4, "Hardware Requirements"](#)
- [Section 11.5, "Configuration Options in WebLogic Server with Oracle RAC"](#)

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

11.1 Overview of Oracle Real Application Clusters

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 host the database instances are linked by a high-speed interconnect to form the cluster. The interconnect is a physical network used as a means of communication between the nodes of the cluster. Cluster functionality is provided by the operating system or compatible third party clustering software.

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.

11.2 Software Requirements

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. 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 Veritas Cluster File System. Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

Note: See "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.

11.3 JDBC Driver Requirements

To use WebLogic Server with Oracle RAC, your WebLogic JDBC data sources must use the Oracle JDBC Thin driver 11g to create database connections.

11.4 Hardware Requirements

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

11.4.1 WebLogic Server Cluster

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

11.4.2 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.

11.4.3 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.

11.5 Configuration Options in WebLogic Server with Oracle RAC

When using WebLogic Server 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. The following sections describe configuration options and requirements:

- [Section 11.5.1, "Choosing a WebLogic Server Configuration for Use with Oracle RAC"](#)
- [Section 11.5.2, "Validating Connections when using WebLogic Server with Oracle RAC"](#)
- [Section 11.5.3, "Additional Considerations When Using WebLogic Server with Oracle RAC"](#)

11.5.1 Choosing a WebLogic Server Configuration for Use with Oracle RAC

Consider the following alternatives:

- Using GridLink data sources, see *Oracle® Fusion Middleware Licensing Information*.
- To connect to multiple Oracle RAC instances when using global transactions (XA), Oracle recommends the use of transaction-aware WebLogic JDBC multi data sources, which support failover and load balancing, to connect to the Oracle RAC nodes. For more information see [Section C.6, "Using Multi Data Sources with Global Transactions."](#)
- To connect to multiple Oracle RAC 11g instances when not using XA, Oracle recommends the use of (non-transaction-aware) multi data sources to connect to the Oracle RAC nodes. Use the standard multi data source configuration, which supports failover and load balancing. For more information see [Section C.7, "Using Multi Data Sources without Global Transactions."](#)

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

Table 11–1 Choosing Configurations to Use with Oracle RAC

Requires Load Balancing?	Requires Failover?	Requires Global Transactions (XA)?	Uses Oracle RAC Services	See...
Yes	Yes	Yes	Yes	Section 4, "Using GridLink Data Sources"
Yes	Yes	Yes	No	Section C.6, "Using Multi Data Sources with Global Transactions"
Yes	Yes	Yes	Yes	Section C.8, "Configuring Connections to Services on Oracle RAC Nodes"
Yes	Yes	No	Yes	Section C.8, "Configuring Connections to Services on Oracle RAC Nodes"
Yes	Yes	No	No	Section C.7, "Using Multi Data Sources without Global Transactions"

WebLogic supports the use of Oracle Data Guard with Multi Data Source and AGL. When used with a Multi Data Source, the following limitations exist:

- Only the failover policy is supported.
- Only one RAC instance is allowed in the primary data center. A single generic data source that is a member of the Multi Data Source is configured for the primary data center. If SCAN is used, an `INSTANCE_NAME` must also be specified.
- For each standby instance, a generic data source that is a member of the Multi Data Source must be configured. If SCAN is used, an `INSTANCE_NAME` must also be specified for each instance. No member of the Multi Data Source can represent more than one instance in a RAC cluster.

11.5.2 Validating Connections when using WebLogic Server with Oracle RAC

Applications can use the JDBC 4.0 `Connection.isValid` API to verify connection viability.

Note: WebLogic Server does not support
`oracle.ucp.jdbc.ValidConnection.isValid` or
`oracle.ucp.jdbc.ValidConnection.setInvalid`.

11.5.3 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 AGL data sources 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 AGL or Multi Data Source. 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.

Using JDBC Drivers with WebLogic Server

This chapter describes how to set up and use JDBC drivers with WebLogic Server.

- [Section 12.1, "JDBC Driver Support"](#)
- [Section 12.2, "JDBC Drivers Installed with WebLogic Server"](#)
- [Section 12.3, "Setting the Environment for a Third-Party JDBC Driver"](#)
- [Section 12.4, "Globalization Support for the Oracle 11g Thin Driver"](#)
- [Section 12.5, "Using the Oracle Thin Driver in Debug Mode"](#)
- [Section 12.6, "Using the Oracle Thin Driver Over SSL"](#)

12.1 JDBC Driver Support

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

12.2 JDBC Drivers Installed with WebLogic Server

The 11g version of the Oracle Thin driver (`ojdbc6_g.jar` for JDK 6) is installed with Oracle WebLogic Server.

Note: WebLogic-branded DataDirect drivers are also installed with WebLogic Server. See *Using WebLogic-branded DataDirect Drivers* for more information.

In addition to the Oracle Thin Driver, the MySQL 5.0.x (`mysql-connector-java-commercial-5.0.x-bin.jar`) JDBC driver is installed with WebLogic Server.

This driver is installed in the `WL_HOME\server\lib` folder (where `WL_HOME` is the folder where WebLogic Server is installed) with `weblogic.jar`. The manifest in `weblogic.jar` lists 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*.

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

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

Note: WebLogic Server also includes 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>.

12.3 Setting the Environment for a Third-Party JDBC Driver

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

12.4 Globalization Support for the Oracle 11g Thin Driver

For Globalization Support with the 11g version of the Oracle Thin driver, Oracle supplies the `ora118n.jar` file, which replaces `nls_charset.zip`. If you use character sets other than US7ASCII, WE8DEC, WE8ISO8859P1 and UTF8 with CHAR

and NCHAR data in Oracle object types and collections, you must include `orai18n.jar` and `orai18n-mapping.jar` in your `CLASSPATH`.

The `orai18n.jar` and `orai18n-mapping.jar` are included with the WebLogic Server installation in the `WL_HOME\server\ext\jdbc\oracle\11g` 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.

12.5 Using the Oracle Thin Driver in Debug Mode

The `WL_HOME\server\ext\jdbc\oracle\11g` folder includes the `ojdbc6_g.jar` (for JDK 6) and `ojdbc5_g.jar` (for JDK 5) files, which are the versions of the Oracle Thin driver with classes to support debugging and tracing. To use the Oracle Thin driver in debug mode, add the path to these files at the beginning of your `CLASSPATH`.

12.6 Using the Oracle Thin Driver Over SSL

This section describes how to configure the data source in WebLogic Server when using the Oracle Thin driver over SSL.

To configure the data source in WebLogic Server, perform the following steps.

1. In the Administration Console navigation pane, click the plus sign to expand Services.
2. Click the plus sign to expand JDBC.
3. Select Data Sources.
4. On the Summary of JDBC Data Sources page, select New.
5. On the Create a New JDBC Data Source page, provide a JDBC data source name and, optionally, the JNDI name.

Select Oracle as the database type, and the Oracle database driver (Thin XA or Thin) you want to use.

6. Click Next.
7. Select the transaction options, if any. Click Next.
8. On the Connection Properties page, enter the Database Name, the Host Name, the Port, and the Database User Name and password. These values are used to construct a URL for the data source. You will modify this URL on the next page, but you need to enter values now to proceed. Click Next.
9. On the Test Configuration page, make the following changes:
 - a. Verify that the driver Class Name is `oracle.jdbc.xa.client.OracleXADataSource` or `oracle.jdbc.OracleDriver`.
 - b. Modify the URL to use the following syntax and format:

```
URL : jdbc:oracle:thin:@(DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL = tcps)
(HOST = <hostname>) (PORT = <sslport>))) (CONNECT_DATA = (SERVICE_NAME = <service_name>))
(SEcurity=(SSL_SERVER_CERT_DN=\"CN=<.....>\")))
```

`(SECURITY=(SSL_SERVER_CERT_DN=\"CN=<.....>\"))` is optional and is used only for server DN matching.

10. Enter (or accept) the Test Table Name.

11. Optionally, click Test Configuration to test the configuration.
12. Click Finish.

Monitoring WebLogic JDBC Resources

This chapter provides information on 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.

The following sections include details about monitoring JDBC objects:

- [Section 13.1, "Viewing Run-Time Statistics"](#)
- [Section 13.2, "Profile Logging"](#)
- [Section 13.3, "Collecting Profile Information"](#)
- [Section 13.4, "Debugging JDBC Data Sources"](#)

13.1 Viewing Run-Time Statistics

Viewing run-time statistics allows you to monitor the data sources in your WebLogic domain.

13.1.1 Data Source Statistics

You can view run-time statistics for a data source via the Administration Console or through the `JDBCDataSourceRuntimeMBean`. 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 *Oracle WebLogic Server MBean Reference*.

13.1.2 Prepared Statement Cache Statistics

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

13.2 Profile Logging

Prior to WebLogic Server 10.3.6, data source profile records were recorded as WLDF events. To provide better usability and performance, WebLogic Server now 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 *Oracle WebLogic Server MBean Reference*.
- Data accessibility—provides the ability to use common text editors, the WLDF Data Accessor, or the Administration Console. See [Section 13.3.2, "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 [Section 13.3.1, "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 *Oracle WebLogic Server MBean Reference*.

For more information on WebLogic logging services, see:

- "Enable and configure Datasource Profile logs" in *Oracle WebLogic Server Administration Console Help*.
- "Understanding WebLogic Logging Services" in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

13.3 Collecting Profile Information

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

When configuring your data source for profiling, you must specify the interval at which profile data is harvested (`Harvest Frequency Seconds`); if the interval is set to 0, harvesting of data is disabled. See "Configure diagnostic profiling for a JDBC data source" in *Oracle WebLogic Server Administration Console Help*.

13.3.1 Profile Types

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

- [Section 13.3.1.1, "Connection Usage \(PROFILE_TYPE_CONN_USAGE_STR\)"](#)
- [Section 13.3.1.2, "Connection Reservation Wait \(PROFILE_TYPE_CONN_RESV_WAIT_STR\)"](#)

- [Section 13.3.1.3, "Connection Reservation Failed \(PROFILE_TYPE_CONN_RESV_FAIL_STR\)"](#)
- [Section 13.3.1.4, "Connection Leak \(PROFILE_TYPE_CONN_LEAK_STR\)"](#)
- [Section 13.3.1.5, "Connection Last Usage \(PROFILE_TYPE_CONN_LAST_USAGE_STR\)"](#)
- [Section 13.3.1.6, "Connection Multithreaded Usage \(PROFILE_TYPE_CONN_MT_USAGE_STR\)"](#)
- [Section 13.3.1.7, "Statement Cache Entry \(PROFILE_TYPE_STMT_CACHE_ENTRY_STR\)"](#)
- [Section 13.3.1.8, "Statements Usage \(PROFILE_TYPE_STMT_USAGE_STR\)"](#)
- [Section 13.3.1.9, "Connection Unwrap \(WEBLOGIC.JDBC.CONN.UNWRAP\)"](#)
- [Section 13.3.1.10, "Example Profile Information Record Log"](#)

13.3.1.1 Connection Usage (PROFILE_TYPE_CONN_USAGE_STR)

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

Note: By default, enabling connection usage profiling on its own will not provide a stack trace of the threads using the connections. To obtain this information you must enable profiling of connection leaks in addition to enabling connection. For more information about profiling connection leaks see [Section 13.3.1.4, "Connection Leak \(PROFILE_TYPE_CONN_LEAK_STR\)."](#)

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

13.3.1.2 Connection Reservation Wait (PROFILE_TYPE_CONN_RESV_WAIT_STR)

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

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

13.3.1.3 Connection Reservation Failed (PROFILE_TYPE_CONN_RESV_FAIL_STR)

Enable connection reservation failure profiling to collect information about threads that attempt to reserve a connection from the data source but fail to get that

connection. This profile information can help determine why applications are unable to get connections from the data source even after reserving them. The record contains the following information:

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

13.3.1.4 Connection Leak (PROFILE_TYPE_CONN_LEAK_STR)

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

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

13.3.1.5 Connection Last Usage (PROFILE_TYPE_CONN_LAST_USAGE_STR)

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

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

13.3.1.6 Connection Multithreaded Usage (PROFILE_TYPE_CONN_MT_USAGE_STR)

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

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

13.3.1.7 Statement Cache Entry (PROFILE_TYPE_STMT_CACHE_ENTRY_STR)

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

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

13.3.1.8 Statements Usage (PROFILE_TYPE_STMT_USAGE_STR)

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

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

13.3.1.9 Connection Unwrap (WEBLOGIC.JDBC.CONN.UNWRAP)

Enable statements usage 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.

13.3.1.10 Example Profile Information Record Log

The following is an example profile information record for [Section 13.3.1.8, "Statements Usage \(PROFILE_TYPE_STMT_USAGE_STR\)"](#) from a standard output log.

```
####<JDBC Data Source-0> <WEBLOGIC.JDBC.STMT.USAGE> <0> <java.lang.Exception
    at
    .
    .
    .
    weblogic.servlet.provider.ContainerSupportProviderImpl$WlsRequestExecutor.run(
    ContainerSupportProviderImpl.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`

13.3.2 Accessing Diagnostic Data

You can use one of the following methods to access diagnostic data:

- The WebLogic Administration Console. See:
 - "View and configure logs" in *Oracle WebLogic Server Administration Console Help*.
 - "Monitor Statistics for a JDBC data source" in *Oracle WebLogic Server Administration Console Help*.
- 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.

13.3.3 Callbacks for Monitoring Driver-Level Statistics (Deprecated)

Note: This feature is deprecated in WebLogic Server 10.3.6.0 and may be removed in a future release.

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.

13.4 Debugging JDBC Data Sources

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

13.4.1 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.

13.4.1.1 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.

13.4.1.2 Enable Debugging Using the WebLogic Server Administration Console

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

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

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

13.4.1.3 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.

13.4.1.4 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 13-1](#):

Example 13-1 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.

13.4.2 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.
- `DebugJDBCRAAC` (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.
- `DebugJDBCRMIM` (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 `getJDBCLogFileFileName`). Note that 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.

13.4.3 Setting Debugging for UCP/ONS

For WebLogic Server releases 10.3.6.0 and higher, the package names for UCP and ONS are no longer repackaged which affects debugging for these components.

13.4.3.1 Debugging UCP

Set UCP debugging directly using:

```

oracle.ucp.level = FINEST;
oracle.ucp.jdbc.PoolDataSource = WARNING;

```

13.4.3.2 Debugging ONS

To enable debugging for ONS, use:

```
oracle.ons.debug=true
```

To print output, make the following call:

```
oracle.ons.ONS public static void setLogStream(PrintStream ps, PrintStream ps2);
```

13.4.4 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*.

Managing WebLogic JDBC Resources

This chapter provides information on how to use the Administration Console, command line, JMX programs, or WebLogic Scripting Tool (WLST) scripts to manage the JDBC data sources in your domain.

- [Section 14.1, "Testing Data Sources and Database Connections"](#)
- [Section 14.2, "Managing the Statement Cache for a Data Source"](#)
- [Section 14.3, "Shrinking a Connection Pool"](#)
- [Section 14.4, "Resetting a Connection Pool"](#)
- [Section 14.5, "Suspending a Connection Pool"](#)
- [Section 14.6, "Resuming a Connection Pool"](#)
- [Section 14.7, "Shutting Down a Data Source"](#)
- [Section 14.8, "Starting a Data Source"](#)
- [Section 14.9, "Managing DBMS Network Failures"](#)

14.1 Testing Data Sources and Database Connections

```
JDBCDataSourceRuntimeMBean.testPool
```

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

Allowing 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 [Section 10.2, "Connection Testing Options for a Data Source."](#)

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

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

14.2 Managing the Statement Cache for a Data Source

For each connection in a data source in your system, WebLogic Server creates a statement cache. When a prepared statement or callable statement is used on a connection, WebLogic Server caches the statement so that it can be reused. For more information about the statement cache, see [Section 10.1, "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 Administration Console or you can programmatically clear the statement cache for an *individual* connection.

14.2.1 Clearing the Statement Cache for a Data Source

```
JDBCDataSourceRuntimeMBean.clearStatementCache
```

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

14.2.2 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`.

14.3 Shrinking a Connection Pool

```
JDBCDataSourceRuntimeMBean.shrink
```

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 on the JDBC Data Source: Control page in the Administration Console (see "Shrink the connection pool in a JDBC data source" in *Oracle WebLogic Server Administration Console Help*) or the `shrink()` method on the `JDBCDataSourceRuntimeMBean`. When you shrink a data source, WebLogic Server reduces the number of connections in the pool to the greater of either the `minCapacity` or the number of connections currently in use.

14.4 Resetting a Connection Pool

```
JDBCDataSourceRuntimeMBean.reset
```

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

14.5 Suspending a Connection Pool

```
JDBCDataSourceRuntimeMBean.suspend  
JDBCDataSourceRuntimeMBean.forceSuspend
```

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

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

When you forcibly suspend a data source, 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.

14.6 Resuming a Connection Pool

```
JDBCDataSourceRuntimeMBean.resume
```

To re-enable a data source that you suspended, you can use the Resume option on the JDBC Data Source: Control page in the Administration Console (see "Resume suspended JDBC data sources" in *Oracle WebLogic Server Administration Console Help*) or the `resume()` method on the `JDBCDataSourceRuntimeMBean`. When you resume a data source, WebLogic Server marks the data source as enabled and allows applications to use connections from the data source. If you suspended the data source (not forcibly suspended), all connections are preserved exactly as they were before the data source was suspended. Clients that had reserved a connection before the data source was suspended can continue 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.

14.7 Shutting Down a Data Source

```
JDBCDataSourceRuntimeMBean.shutdown  
JDBCDataSourceRuntimeMBean.forceShutdown
```

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

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

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

14.8 Starting a Data Source

```
JDBCDataSourceRuntimeMBean.start
```

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

14.9 Managing DBMS Network Failures

```
-Dweblogic.resourcepool.max_test_wait_secs=xx
```

where `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 applications to hang for extended periods of time (for example, 10 minutes). If the assigned time period expires, the server instance purges and disables the pool (closes all connections and blocks further reserve attempts) and re-enables the pool as soon as it is possible to reconnect.

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.

Configuring JDBC Application Modules for Deployment

This chapter provides information on how to package and scope enterprise applications.

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

The following sections describe the details of packaged JDBC modules:

- [Section A.1, "Packaging a JDBC Module with an Enterprise Application: Main Steps"](#)
- [Section A.2, "Creating Packaged JDBC Modules"](#)
- [Section A.3, "Referencing a JDBC Module in Java EE Descriptor Files"](#)
- [Section A.4, "Packaging an Enterprise Application with a JDBC Module"](#)
- [Section A.5, "Deploying an Enterprise Application with a JDBC Module"](#)
- [Section A.6, "Getting a Database Connection from a Packaged JDBC Module"](#)

A.1 Packaging a JDBC Module with an Enterprise Application: Main Steps

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

1. Create the module. See [Section A.2, "Creating Packaged JDBC Modules."](#)
2. Add references to the module in all applicable descriptor files. See [Section A.3, "Referencing a JDBC Module in Java EE Descriptor Files."](#)
3. Package all application modules in an EAR. See [Section A.4, "Packaging an Enterprise Application with a JDBC Module."](#)
4. Deploy the application. See [Section A.5, "Deploying an Enterprise Application with a JDBC Module."](#)

A.2 Creating Packaged JDBC Modules

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

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

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

A.2.1 Creating a JDBC Data Source Module Using the Administration Console

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

1. Create a data source as described in [Section 3.3, "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 [Section A.2.7, "Application Scoping for a Packaged JDBC Module."](#)
4. Continue with adding references to the descriptor files in the enterprise application. See [Section A.3, "Referencing a JDBC Module in Java EE Descriptor Files."](#)

A.2.2 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://xmlns.oracle.com/weblogic/jdbc-data-source/1.2/jdbc-data-source.xsd>.
- Uses a file name that ends in `-jdbc.xml`.
- Includes a name element that is unique within the WebLogic domain.

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

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

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

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

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

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

A.2.3 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.

A.2.4 Creating a JDBC Data Source Module

The main sections within a JDBC data source module are:

- `jdbc-driver-params`—includes entries for the JDBC driver used to create database connections, including `url`, `driver-name`, and individual driver property entries. See the `jdbc-data-source.xsd` schema for more valid entries. For an explanation of each element, see "JDBCDataSourceParamsBean" in the *Oracle WebLogic Server MBean Reference*.
- `jdbc-connection-pool-params`—includes entries for connection pool configuration, including connection testing options, statement cache options, and so forth. This element also inherits `connection-pool-params` from the `weblogic-javaee.xsd` schema, including `initial-capacity`, `max-capacity`, and other options common to pooled resources. For more information, see the following:
 - "JDBCConnectionPoolParamsBean" in the *Oracle WebLogic Server MBean Reference*
 - `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 *Oracle WebLogic Server MBean Reference*.

- `jdbc-xa-params`—includes entries for XA database connection handling options, such as `keep-xa-conn-till-tx-complete`, and `xa-transaction-timeout`. For an explanation of each element, see "JDBCXAParamsBean" in the *Oracle WebLogic Server MBean Reference*.

[Example A-1](#) shows an example of a JDBC module for a data source with some typical configuration options.

Example A-1 Sample JDBC 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/00iSWRYu2rZGUwnVo6U=</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>
```

A.2.5 Creating a JDBC Multi Data Source Module

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

- `jndi-name` (required)
- `data-source-list` (required)
- `scope`
- `algorithm-type`
- `connection-pool-failover-callback-handler`
- `failover-request-if-busy`

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

[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>
```

A.2.6 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 *WebLogic Scripting Tool Command Reference*.

A.2.6.1 Deploying JDBC Modules to New Domains

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

- Override the old encrypted password within a deployment plan that includes a password that was encrypted specifically for the new domain. See "Update a deployment plan" in *Oracle WebLogic Server Administration Console Help*.
- Re-encrypt the passwords for your new domain. See [Section A.2.6, "Encrypting Database Passwords in a JDBC Module."](#)

A.2.7 Application Scoping for a Packaged JDBC Module

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

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

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


```

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

```

A.3.2 Packaged JDBC Module References in Other Descriptors

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

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

```

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

```

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

```

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

```

- In the WebLogic-specific descriptor files, such as `weblogic-ear.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.

A.4 Packaging an Enterprise Application with a JDBC Module

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

A.5 Deploying an Enterprise Application with a JDBC Module

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

A.6 Getting a Database Connection from a Packaged JDBC Module

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

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

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

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

WebLogic Server 10.3.6 Support for Oracle 12c Database

This chapter provides information on how to configure WebLogic Server Release 10.3.6 to run with an Oracle 12c database.

- [Section B.1, "Using an Oracle 12c Database with WebLogic Server 10.3.6"](#)
- [Section B.2, "JDBC 4.1 Support for JDK 7"](#)
- [Section B.3, "Using Application Continuity"](#)
- [Section B.4, "Global Database Services"](#)
- [Section B.5, "Container Database with Pluggable Databases"](#)
- [Section B.6, "Automatic ONS Listener Support"](#)
- [Section B.7, "Database Resident Connection Pooling Support"](#)

B.1 Using an Oracle 12c Database with WebLogic Server 10.3.6

The following sections provide information on how to prepare and use an Oracle 12c database with WebLogic Server 10.3.6:

- [Section B.1.1, "Using 11g Drivers with Oracle 12c Database"](#)
- [Section B.1.2, "Using 12c Drivers with Oracle 12c Database"](#)

B.1.1 Using 11g Drivers with Oracle 12c Database

The simplest integration of WebLogic Server 10.3.6 with a 12c database is to use the 11g (specifically 11.2.0.3) driver jar files included in your WebLogic Server 10.3.6 distribution. There are no known problems or upgrade issues when using 11g drivers with a 12c database. Please note the following:

- The Pluggable Database (PDB) feature can be used without any new client jar files or WebLogic Server changes. However, the `ALTER SESSION SET CONTAINER` feature to dynamically change containers from the client cannot be used.
- Application Continuity is not supported.
- Database resident connection pool (DRCP) and ONS auto configuration are not supported.
- Global database service (GDS) can be used with an Active GridLink data source.
- JDBC 4.1 methods are not supported.

B.1.2 Using 12c Drivers with Oracle 12c Database

The following sections provides information on how to prepare and use 12c drivers to run with WebLogic Server 10.3.6 and an Oracle 12c database:

- [Section B.1.2.1, "Required Oracle 12c Driver Files"](#)
- [Section B.1.2.2, "Download Oracle 12c Database Files"](#)
- [Section B.1.2.3, "Update the WebLogic Server CLASSPATH"](#)
- [Section B.1.2.4, "Other Considerations"](#)

B.1.2.1 Required Oracle 12c Driver Files

This section lists the files required to use an Oracle 12c driver with WebLogic Server 10.3.6. These files are installed under the 12c database \$ORACLE_HOME directory. It is necessary to install either the Client package or the Database package.

Note: These jar files must be added to the CLASSPATH used for running WebLogic Server at the head of the CLASSPATH. They must come before all of the 11g client jar files.

Select one of the following ojdbc files :

- jdbc/lib/ojdbc7.jar
- jdbc/lib/ojdbc7_g.jar
- jdbc/lib/ojdbc7dms.jar
- jdbc/lib/ojdbc6.jar
- jdbc/lib/ojdbc6_g.jar
- jdbc/lib/ojdbc6dms.jar

The following table lists additional required driver files:

Table B-1 Additional 12c Driver Files

File	Description
ucp/lib/ucp.jar	Universal Connection Pool
opmn/lib/ons.jar	Oracle Network Server client
jlib/orai18n.jar	I18N support
jlib/orai18n-mapping.jar	I18N support
jlib/oraclepki.jar	Oracle Wallet support
jlib/osdt_cert.jar	Oracle Wallet support
jlib/osdt_core.jar	Oracle Wallet support

The following table lists additional files for SQL XML support:

Table B-2 Additional Files for SQL XML Support

File	Description
rdbms/jlib/xdm.jar	SQLXML support
lib/ucp.jar	

Table B-2 (Cont.) Additional Files for SQL XML Support

File	Description
lib/xmlparserv2_sans_jaxp_services.jar	SQLXML support
jlib/orai18n-collation.jar	SQLXML support

B.1.2.2 Download Oracle 12c Database Files

The 12c Oracle database jar files are not shipped with this version of WebLogic Server. Oracle recommends that you do a custom install of the client kit for a minimal installation. See

<http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html>.

Under **Oracle Database 12c Release 1**, select the **See All** link for your OS platform. For a minimal install, under the **Oracle Database 12c Release 1 Client** heading, select the proper zip file and download it. Unzip the file and run the installer. Select **Custom**, then select the **Oracle JDBC/Thin interfaces**, **Oracle Net listener**, and **Oracle Advanced security check** boxes.

B.1.2.3 Update the WebLogic Server CLASSPATH

To use an Oracle 12c database and Oracle 12c JDBC driver, you must update the CLASSPATH in your WebLogic Server environment. Prepend the required files specified in [Section B.1.2.1, "Required Oracle 12c Driver Files"](#) to the CLASSPATH (before the 11.2.0.3 Driver jar files).

The following code sample outlines a simple shell script that updates the CLASSPATH of your WebLogic environment. Make sure `ORACLE_HOME` is set appropriately. This example assumes the `SEP` variable is set to the correct CLASSPATH separator (";" or ":") for your platform.

```
#!/bin/sh

# source this file in to add the new 12c jar files at the beginning of the
# CLASSPATH

case "`uname`" in
*CYGWIN*)
SEP=";"
;;
Windows_NT)
SEP=";"
;;
*)
SEP=":"
;;
esac
# For external, just use ORACLE_HOME;
dir=${ORACLE_HOME:?}

# We need one of the following
#jdbc/lib/ojdbc7.jar
#jdbc/lib/ojdbc7_g.jar
#jdbc/lib/ojdbc7dms.jar
#jdbc/lib/ojdbc6.jar
#jdbc/lib/ojdbc6_g.jar
#jdbc/lib/ojdbc6dms.jar
```

```

if [ "$1" = "" ]
then
ojdbc=ojdbc7.jar
else
ojdbc="$1"
fi
case "$ojdbc" in
ojdbc7.jar|ojdbc7_g.jar|ojdbc7dms.jar|ojdbc6.jar|ojdbc6_g.jar|ojdbc6dms.jar)
ojdbc=jdbc/lib/$ojdbc
;;
*)
echo "Invalid argument - must be ojdbc7.jar|ojdbc7_
g.jar|ojdbc7dms.jar|ojdbc6.jar|ojdbc6_g.jar|ojdbc6dms.jar"
exit 1
;;
esac
CLASSPATH="${dir}/${ojdbc}${SEP}$CLASSPATH"
CLASSPATH="${dir}/ucp/lib/ucp.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/opmn/lib/ons.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/orai18n.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/orai18n-mapping.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/oraclepki.jar ${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/osdt_cert.jar ${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/osdt_core.jar${SEP}$CLASSPATH"
# SQL XML support
CLASSPATH="${dir}/lib/xmlparserv2_sans_jaxp_services.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/jlib/orai18n-collation.jar${SEP}$CLASSPATH"
CLASSPATH="${dir}/rdbms/jlib/xdm.jar${SEP}$CLASSPATH"

```

For example, save this script in your environment with the name `setdb12c_jars.sh`. Then run script with the `ojdbc6dms.jar`:

```
./setdb12c_jars.sh ojdbc6dms.jar
```

B.1.2.4 Other Considerations

This section provides additional information when using WebLogic Server 10.3.6 with an Oracle 12c database:

- WebLogic Server 10.3.6 does not support using the `aqapi.jar` file from the 12c database distribution and does not support using AQ JMS with the 12c database.
- WebLogic Server 10.3.6 supports Application continuity when using both the 12c client jar files and 12c database with an Active GridLink data source. See [Section B.3, "Using Application Continuity."](#)
- Oracle supports running the 12c client jar files with an 11g database.
- If you are running with Active GridLink, you must add the `-Doracle.ucp.PreWLS1212Compatible=true` system property to the server command line when running with 12c client jar files.
- 12c client jars contain several changes in local transaction processing to maintain compliance with the JDBC specification. If `setAutoCommit(true)` is called and a local transaction exists, the transaction is automatically committed (in earlier releases, no action is taken). If `setAutoCommit(true)` is set and `commit` is called, an exception is thrown (in earlier releases, no action is taken). It is possible that existing applications will encounter these situations and it may be difficult to immediately change the software. Add the

-Doracle.jdbc.autoCommitSpecCompliant=false system property to the server command line to preserve the legacy behavior existing applications.

- JDBC 4.1 methods are supported when running with `ojdbc7*.jar` files

B.2 JDBC 4.1 Support for JDK 7

This release of 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 with the Oracle thin client, you must use the `ojdbc7.jar`. If the methods are called for a driver that does not support JDBC 4.1, a `java.lang.AbstractMethodError` is thrown. See "JDBC™ 4.1 Specification" at http://download.oracle.com/otndocs/jcp/jdbc-4_1-mrel-spec/index.html.

Note: WebLogic Server currently does not support the `java.sql.driver` interfaces required to use the Java SE 7 `getParentLogger` method. See <http://docs.oracle.com/javase/7/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`.

B.3 Using Application Continuity

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 a warning is ignored, users may unintentionally purchase items twice or submit multiple payments for the same invoice.

Application Continuity (also referred to as Replay) is a general purpose, application-independent infrastructure for Active GridLink data sources that enables the recovery of work from an application perspective and masks many system, communication, and hardware failures. The 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 sections provide information on how to configure and use Application Continuity:

- [Section B.3.1, "How Application Continuity Works"](#)
- [Section B.3.2, "Requirements and Considerations"](#)
- [Section B.3.3, "Configuring Application Continuity"](#)

B.3.1 How Application Continuity Works

Following any outage that is due to a loss of database service, planned or unplanned, Application Continuity 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.

B.3.2 Requirements and Considerations

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

- Requires an Oracle 12c JDBC Driver and Database. See [Section B.1.2, "Using 12c Drivers with Oracle 12c Database."](#)
- For this release, Application Continuity only supports read and write transactions. XA transactions are not supported. See [Section B.3.3.1, "Selecting the Driver for Application Continuity."](#)

Note: Remember to set `autocommit=FALSE` to prevent breaking the transaction semantics and disabling Application Continuity 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.
- Application Continuity works by storing intermediate results in memory. An application may run slower and require significantly more memory than running without the feature.
- Using the WebLogic statement cache is not supported. If the WebLogic statement cache is configured, the cache is cleared every time the connection is closed.
- There are additional limitations and exceptions to the Application Continuity 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 datasource 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
```

B.3.3 Configuring Application Continuity

The following sections provide information on how to implement Application Continuity in your environment:

- [Section B.3.3.1, "Selecting the Driver for Application Continuity"](#)
- [Section B.3.3.2, "Using a Connection Callback"](#)
- [Section B.3.3.3, "Disabling Application Continuity for a Connection"](#)
- [Section B.3.3.4, "Configuring Logging for Application Continuity"](#)

B.3.3.1 Selecting the Driver for Application Continuity

The administration console does not have an entry for a replay driver. To configure the JDBC replay data source, you must manually replace the existing Oracle data source class with the replay data source class

```
oracle.jdbc.replay.OracleDataSourceImpl.
```

From the WebLogic Server Administration console:

1. From the WebLogic Server Administration Home page, select **Configure GridLink for RAC Data Source**.
2. Click **New** and select **GridLink Data Source**.
3. In **Database Driver**, select **Oracle's Driver (Thin) for GridLink Connections Versions: 11 and later**.
4. Finish creating the data source without targeting to a server.
5. Select and edit the new data source.
6. From the **Configuration: Connection Pool** tab, replace the value of the **Driver Class Name** attribute with **oracle.jdbc.replay.OracleDataSourceImpl**.
7. Click **Save**.
8. Select the **Targets** tab.
9. Select the target server(s).
10. Click **Save**.

Make sure the **Database Type** is **Oracle**. See [Section B.3.2, "Requirements and Considerations."](#)

B.3.3.2 Using a Connection Callback

The following sections provide information on how to use a Connection Callback:

- [Section B.3.3.2.1, "Create an Initialization Callback"](#)
- [Section B.3.3.2.2, "Registering an Initialization Callback"](#)
- [Section B.3.3.2.3, "Unregister an Initialization Callback"](#)

B.3.3.2.1 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 of the same

callback at run time and at replay ensures 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).

The following example demonstrates a simple initialization callback implementation:

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

B.3.3.2 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 MyConnectionInitializatio
nCallback();
((WLDataSource)ds).registerConnectionInitializationCallback(callback);
. . .
```

The callback can also be registered by entering the callback class in the **Connection Initialization Callback** attribute on the Oracle tab for a data source in the Administration Console.

B.3.3.2.3 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();
. . .
```

B.3.3.3 Disabling Application Continuity for a Connection

You can disable Application Continuity on a per-connection basis using the following:

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

```
}
. . .
```

B.3.3.4 Configuring Logging for Application Continuity

To enable logging of Application Continuity 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. The following is an example of a configuration file that uses the SimpleFormatter and sets the logging level to FINEST:

```
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
#OR - use other formatters like the ones below
#java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter
#java.util.logging.ConsoleHandler.formatter = oracle.ucp.util.logging.UCPFormatter

#OR - use FileHandler instead of ConsoleHandler
#handlers = java.util.logging.FileHandler
#java.util.logging.FileHandler.pattern = replay.log
#java.util.logging.FileHandler.limit = 3000000000
#java.util.logging.FileHandler.count = 1
#java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
oracle.jdbc.internal.replay.level attribute=FINEST
```

See *Using Logging Services for Application Logging for Oracle WebLogic Server*.

B.4 Global Database 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.

The following sections provide information on requirements and configuration for GDS in WebLogic Server:

- [Section B.4.1, "Requirements and Considerations"](#)
- [Section B.4.2, "Creating a GridLink DataSource for GDS Connectivity"](#)

B.4.1 Requirements and Considerations

The following section provides requirements and considerations when using Global Database Services in WebLogic Server:

- Requires an Oracle 12c JDBC Driver and Database. See [Section B.1, "Using an Oracle 12c Database with WebLogic Server 10.3.6."](#)
- It is not possible to use a single SCAN address to replace multiple Global Service manger (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 datasource configuration, one datasource must be defined for the update service and another datasource defined for the read-only service.
- Your application must be written so that update operations are processed by the update datasource and read-only operations are processed by the read-only datasource.

B.4.2 Creating a GridLink DataSource for GDS Connectivity

Use the Administration Console to create a GridLink datasource that uses a modified URL to provide GDS connectivity. See "Create JDBC GridLink data sources" in the *Oracle WebLogic Server Administration Console Help*.

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 Managers
- 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)))
```

B.5 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 Oracle Pluggable Databases" in the *Oracle Database Administrator's Guide*.

- [Section B.5.1, "Creating Service for PDB Access"](#)
- [Section B.5.2, "Setting the PDB using JDBC"](#)

B.5.1 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', 'replaytest_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 Application Continuity, 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;
/
```

B.5.2 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.

Executing `ALTER SESSION SET CONTAINER` is not supported from JDBC in WebLogic Server 10.3.6.

B.6 Automatic ONS Listener Support

In WebLogic Server 10.3.6, automatic ONS listeners are not supported. The ONS configuration information must be specified in the Oracle parameters for the datasource.

B.7 Database Resident Connection Pooling Support

In WebLogic Server 10.3.6, Database Resident Connection Pooling (DRCP) is not supported.

Using Multi Data Sources with Oracle RAC

This chapter provides information on how to configure and use multi data sources when using Oracle Real Application Clusters (RAC) with WebLogic Server. Oracle continues to support multi data source configurations for legacy application environments using RAC.

- [Section C.1, "Overview of Oracle Real Application Clusters"](#)
- [Section C.2, "Software Requirements"](#)
- [Section C.3, "JDBC Driver Requirements"](#)
- [Section C.4, "Hardware Requirements"](#)
- [Section C.5, "Configuring Multi Data Sources with Oracle RAC"](#)
- [Section C.6, "Using Multi Data Sources with Global Transactions"](#)
- [Section C.7, "Using Multi Data Sources without Global Transactions"](#)
- [Section C.8, "Configuring Connections to Services on Oracle RAC Nodes"](#)
- [Section C.9, "XA Considerations and Limitations when using multi Data Sources with Oracle RAC"](#)
- [Section C.10, "JDBC Store Recovery with Oracle RAC"](#)

Both Oracle RAC and WebLogic Server are complex systems. To use them together requires specific configuration on both systems, as well as clustering software and a shared storage solution. This 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.

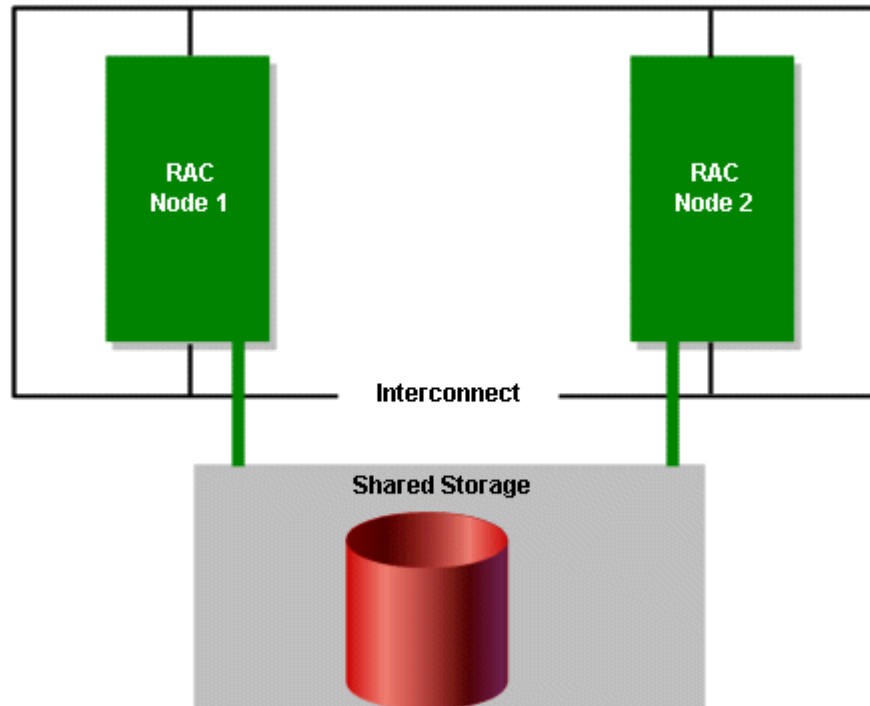
Note: Oracle recommends using GridLink data sources when developing new Oracle RAC applications and when legacy applications do not use multi data sources. See [Section 11, "Using WebLogic Server with Oracle RAC."](#)

C.1 Overview of Oracle Real Application Clusters

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 host the database instances are linked by a high-speed interconnect to form the

cluster. The interconnect is a physical network used as a means of communication between the nodes of the cluster. Cluster functionality is provided by the operating system or compatible third party clustering software. [Figure C-1](#) shows an Oracle RAC configuration.

Figure C-1 Oracle Real Application Clusters Configuration



Oracle RAC offers features in the following areas:

- [Section C.1.1, "Oracle RAC Scalability with WebLogic Server Multi Data Sources."](#)
- [Section C.1.2, "Oracle RAC Availability with WebLogic Server Multi Data Sources."](#)
- [Section C.1.3, "Oracle RAC Load Balancing with WebLogic Server Multi Data Sources."](#)

C.1.1 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.

C.1.2 Oracle RAC Availability with WebLogic Server Multi Data Sources

A multi data source provides an ordered list of data sources to use to satisfy connection requests. Normally, every connection request to this kind of multi data source is served by the first data source in the list. If a database connection test fails and the connection cannot be replaced, or if the data source is suspended, a connection is sought sequentially from the next data source on the list. See [Section 5.3.1, "Failover"](#) and [Section C.5.3.1, "Multi Data Source-Managed Failover and Load Balancing."](#)

C.1.3 Oracle RAC Load Balancing with WebLogic Server Multi Data Sources

Multi data sources provide load balancing for XA and non-XA environments. The generic data sources that form a multi data source are accessed using a round-robin scheme. When switching connections, WebLogic Server selects a connection from the next generic data source in the order listed. For more information about using multi data sources with Oracle RAC, see [Section C, "Using Multi Data Sources with Oracle RAC."](#)

C.2 Software Requirements

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 Veritas Cluster File System. Note that some clustering software includes a file storage solution, in which case additional shared storage software is not required.

Note: See "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.

C.3 JDBC Driver Requirements

To use WebLogic Server with Oracle RAC, your WebLogic JDBC data sources must use the Oracle JDBC Thin driver 11g to create database connections.

C.4 Hardware Requirements

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

C.4.1 WebLogic Server Cluster

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

C.4.2 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.

C.4.3 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.

C.5 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. The following sections describe configuration options and requirements:

- [Section C.5.1, "Choosing a Multi Data Source Configuration for Use with Oracle RAC"](#)
- [Section C.5.2, "Configuring Multi Data Sources for use with Oracle RAC"](#)
- [Section C.5.3, "Configuration Considerations for Failover"](#)
- [Section C.5.4, "Configuring the Listener Process for Each Oracle RAC Instance"](#)
- [Section C.5.5, "Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled"](#)
- [Section C.5.6, "Additional Configuration Considerations"](#)

C.5.1 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 11g instances when using global transactions (XA), see [Section C.6, "Using Multi Data Sources with Global Transactions."](#)
- To connect to multiple Oracle RAC 11g instances when not using XA, see [Section C.7, "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 [Section C.8, "Configuring Connections to Services on Oracle RAC Nodes."](#)

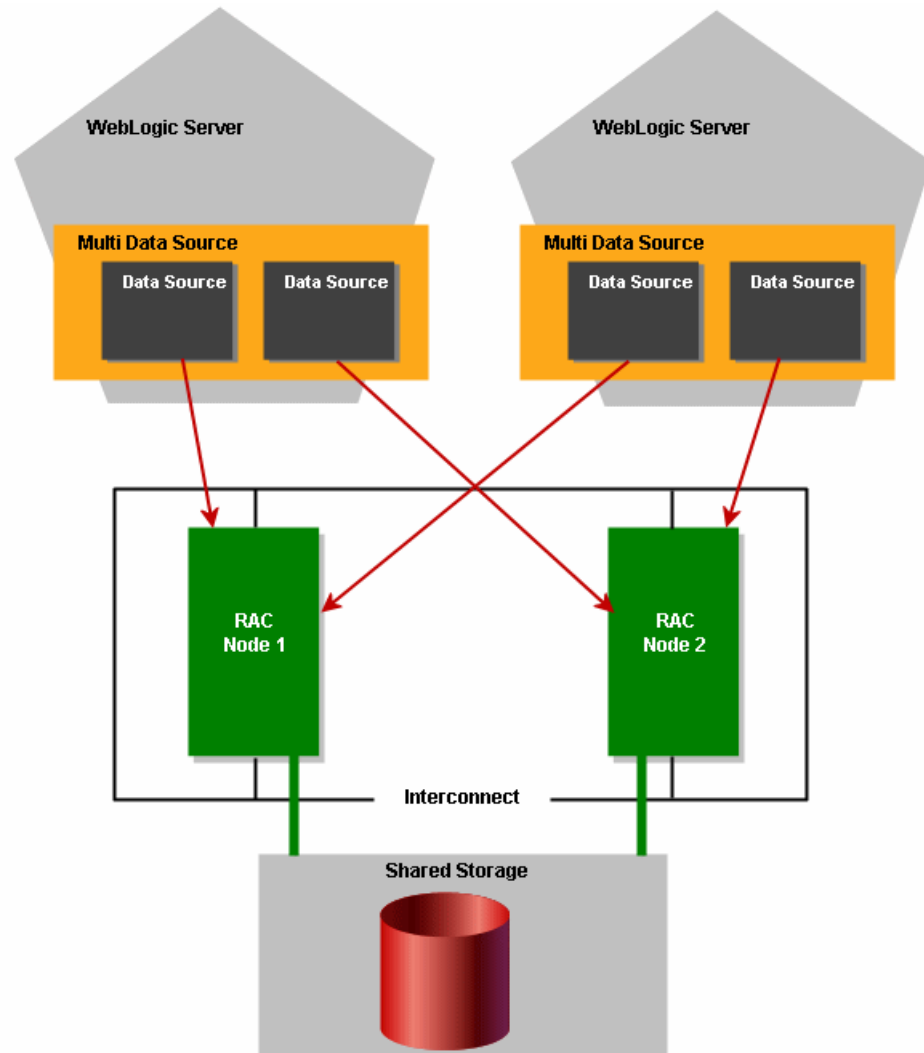
C.5.2 Configuring Multi Data Sources for use with Oracle RAC

To connect WebLogic Server to multiple Oracle RAC nodes using multi data sources, first configure a JDBC 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 the data sources to it.

Figure C-2 shows a typical multi data source configuration.

Figure C-2 Multi Data Source Configuration



You can use the Administration Console or any other means that you prefer to configure your domain, such as the WebLogic Scripting Tool (WLST) or a JMX program. For information about configuring a WebLogic JDBC multi data source see [Section 5, "Configuring JDBC Multi Data Sources."](#) For information on how to configure the data sources in a multi data source to connect to services running on Oracle RAC nodes, see [Section C.8, "Configuring Connections to Services on Oracle RAC Nodes."](#)

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

C.5.2.1 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 data sources; with the High-Availability option, connection requests are served by the first available pool in the list. When a data source becomes defunct, connection requests are served by the next data source in the list.
- `FailoverRequestIfBusy="true"`
With the Failover algorithm, this attribute enables failover when all connections in a data source are in use.
- `TestFrequencySeconds="120"`
This attribute controls the frequency at which WebLogic Server checks the health of data sources previously marked as unhealthy to see if connections can be recreated and if the data source can be re-enabled. For more details see [Section 5, "Configuring JDBC Multi Data Sources."](#)
For fast failover of Oracle RAC nodes, set this value to a smaller interval, for example, 10 (seconds).

C.5.3 Configuration Considerations for Failover

Consider the following information when configuring for failover.

C.5.3.1 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 [Section 5.5, "Multi Data Source Failover Enhancements."](#)

With this configuration, pictured in [Figure C-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 data source and routes connection requests to other 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 data source. When WebLogic Server is successful in recreating the connections, it next re-enables the data source and begins routing connection requests to the data source again. Because of the connection request routing and automatic health checking features, there is minimal delay in satisfying connection requests after a failure.

C.5.3.2 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`.

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

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

```
<jdbc-data-source xmlns="http://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-xa-params>
    ...
    <xa-retry-duration-seconds>300</xa-retry-duration-seconds>
  </jdbc-xa-params>
</jdbc-data-source>
```

Use the following formula to determine the value for `XARetryDurationSeconds`:

`XARetryDurationSeconds` = (longest transaction timeout for transactions that use connections from the data source) + (delay before XIDs are available on all 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 Administration Console, use the following steps:

1. If you have not already done so, in the Change Center of the Administration Console, click **Lock & Edit**.
2. In the **Domain Structure** tree, expand **Services > JDBC**, then select **Data Sources**.
3. On the Summary of Data Sources page, click the data source name.
4. Select the **Configuration: Connection Pool** tab.
5. Scroll down and click **Advanced** to show the advanced connection pool options.
6. Update XA Retry Duration and XA Retry Interval.

7. Click **Save**.

Optionally, you can use WebLogic Scripting Tool (WLST) or a JMX program.

C.5.3.3 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.

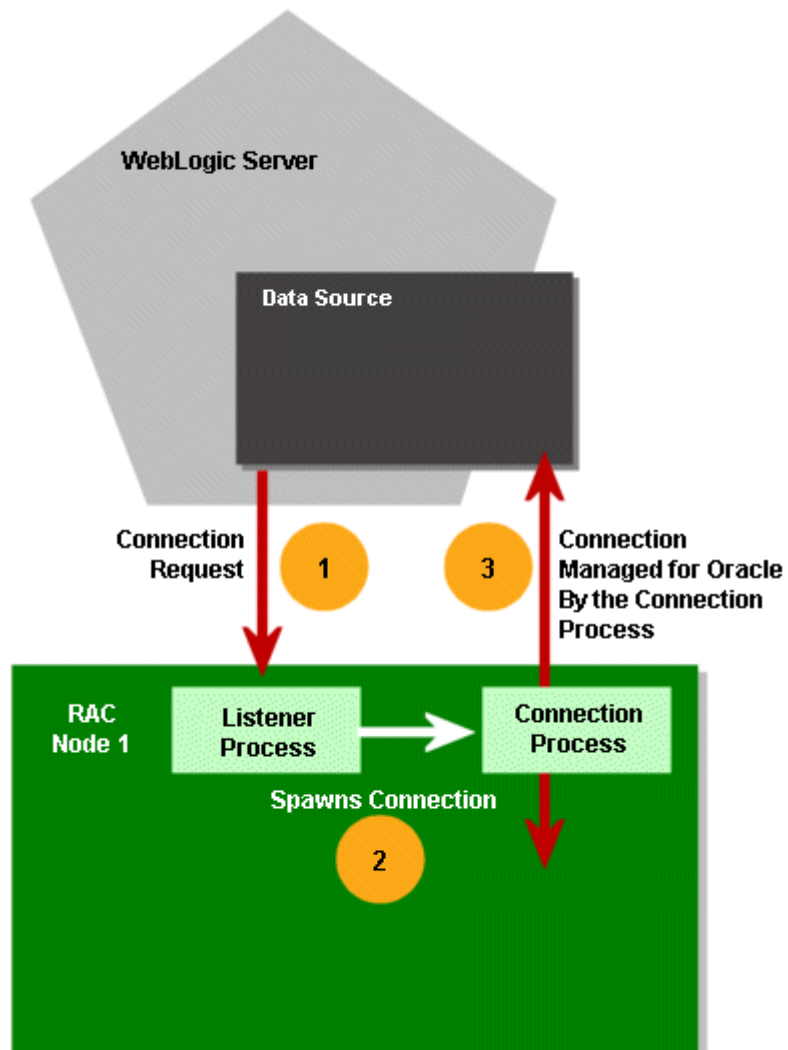
C.5.4 Configuring the Listener Process for Each Oracle RAC Instance

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

When a 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 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 C-3](#) shows the Oracle listener process functionality.

Figure C-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. WLS 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 data sources in the multi data source. See [Section C.5.5, "Configuring Multi Data Sources When Remote Listeners are Enabled or Disabled."](#)

C.5.5 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 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 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)))
```

C.5.6 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 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) (PO
```

```
RT=1521)) (CONNECT_DATA=(SERVICE_
NAME=orcl.us.myco.com) (INSTANCE_NAME=orcl1))
```

The following code snippet provides an example data source configuration:

```
<url>jdbc:oracle:thin:@(DESCRIPTION=(enable=broken) (ADDRESS_
LIST=(ADDRESS=(PROTOCOL=TCP) (HOST=node1-vip.us.myco.com) (PORT=1521))) (CONNECT_
DATA=(SERVICE_NAME=orcl.us.myco.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>
```

C.6 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 operation is retried until the retry duration has expired. If there is a failure after PREPARE the transaction is failed over to another instance.

- [Section C.6.1, "Rules for Data Sources within a Multi Data Source Using Global Transactions"](#)
- [Section C.6.2, "Required Attributes of Data Sources within a Multi Data Source Using Global Transactions"](#)
- [Section C.6.3, "Sample Configuration Code"](#)

C.6.1 Rules for Data Sources within a Multi Data Source Using Global Transactions

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

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

- NewXAConnForCommit
- RollbackLocalTxUponConnClose
- RecoverOnlyOnce
- KeepLogicalConnOpenOnRelease

Note: If you are not using 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 [Section C.7, "Using Multi Data Sources without Global Transactions."](#)

C.6.2 Required Attributes of Data Sources within a Multi Data Source Using Global Transactions

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

- Oracle JDBC Thin driver 11g. For example:

```
<url>jdbc:oracle:thin:@lcgsol24:1521:SNRAC1</url>
<driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
```
- `KeepXAConnTillTxComplete="true"`
 - Forces the data source to reserve a physical database connection and provide the same connection to an application throughout transaction processing until the distributed transaction is complete.
 - Required for proper transaction processing with Oracle RAC.
- `XARetryDurationSeconds="300"`
 - Enables the WebLogic Server transaction manager to retry XA recover, commit, and rollback calls for the specified amount of time.
- `TestConnectionsOnReserve="true"`
 - Enables testing of a database connection when an application reserves a connection from the data source. See [Section 5.4.1, "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 [Section 10.2, "Connection Testing Options for a Data Source."](#)

C.6.3 Sample Configuration Code

Sample configuration code for a multi data source and two associated 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:@lcgsol24:1521: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:@lcqsol25:1521:SNRAC2</url>
    <driver-name>oracle.jdbc.xa.client.OracleXADataSource</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
    <profile-type>0</profile-type>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <jndi-name>oracleRACXAJndiName2</jndi-name>
    <global-transactions-protocol>TwoPhaseCommit
    </global-transactions-protocol>
  </jdbc-data-source-params>
  <jdbc-xa-params>
    <keep-xa-conn-till-tx-complete>true</keep-xa-conn-till-tx-complete>
    <xa-end-only-once>true</xa-end-only-once>
    <xa-set-transaction-timeout>true</xa-set-transaction-timeout>
    <xa-transaction-timeout>120</xa-transaction-timeout>
    <xa-retry-duration-seconds>300</xa-retry-duration-seconds>
  </jdbc-xa-params>
</jdbc-data-source>

```

```

<jdbc-data-source xmlns="http://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>

```

C.7 Using Multi Data Sources without Global Transactions

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

- [Section C.7.1, "Attributes of Data Sources within a Multi Data Source Not Using Global Transactions"](#)
-

C.7.1 Attributes of Data Sources within a Multi Data Source Not Using Global Transactions

Data sources must have the following attributes:

- Oracle JDBC Thin driver 11g. For example:


```

<url>jdbc:oracle:thin:@lcqsol24:1521:SNRAC1</url>
<driver-oracle.jdbc.OracleDriver</driver-name>

```
- TestConnectionsOnReserve="true"
 - Enables testing of a database connection when an application reserves a connection from the data source. [Section 5.4.1, "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 [Section 10.2, "Connection Testing Options for a Data Source."](#)

C.7.2 Sample Configuration Code

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

```

<jdbc-data-source xmlns="http://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:@lcgsol24:1521: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:@lcgsol25:1521: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.

C.8 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 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 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.

[Section C.8.1, "Configuring a Data Source to Connect to a Service,"](#) describes special considerations for configuring these data sources. [Section C.8.2, "Service Connection Configurations,"](#) shows example configurations for either load balancing or workload management.

C.8.1 Configuring a Data Source to Connect to a Service

You configure a data source to connect to a service running on an Oracle RAC node in the same way as you configure any data source (using WLST, the Administration 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 data source even if it can't connect to the service on the node. Without setting this option to 0, data source creation will fail and the server may fail to boot normally.

In the Administration Console, edit the data source after creating it, and set **Initial Capacity** to 0.

- Oracle JDBC Thin (or Thin XA) driver for 11g. 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 Administration 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 Administration 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.

Notes: The SERVICE_NAME must be the same for all data sources in a given multi data source.

Specify a different HOST NAME and/or port for each data source in a given multi data source.

- When specifying `max-capacity` (**Maximum Capacity** in the Administration 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 data sources. See [Section C.8.3, "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 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 data source). At that point, the second 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 data source is connected will experience more use than the remaining nodes on which the service is running.

C.8.2 Service Connection Configurations

You can design your configuration to provide:

- [Section C.8.2.1, "Workload Management"](#)
- [Section C.8.2.2, "Load Balancing"](#)

C.8.2.1 Workload Management

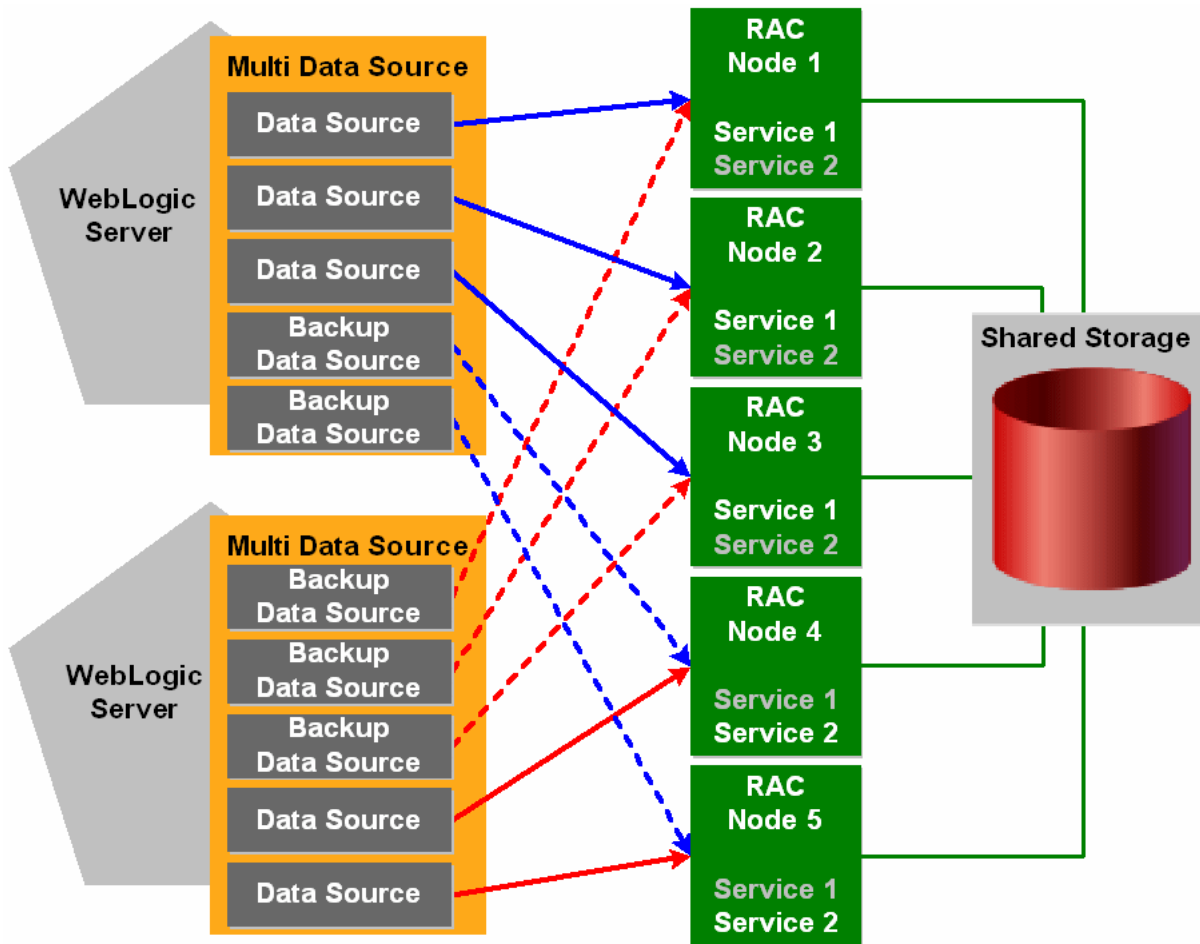
In a workload management configuration, each multi data source has one data source configured for a given service on each Oracle RAC node, regardless of whether the service you are connecting to is active or inactive on a given Oracle RAC node. This lets you quickly start an inactive service on a node 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 data source detects that the service is now active, and the data source will then start making connections to that node as needed. When you stop a service on a given node, the associated data source can no longer make connections to that node, and will become inactive until the service is restarted on that node.

The WLS data source performs connection testing. This lets the data source adjust to changes in the topology of the Oracle RAC configuration. The 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 C-4 Workload Management using Multi Data Sources



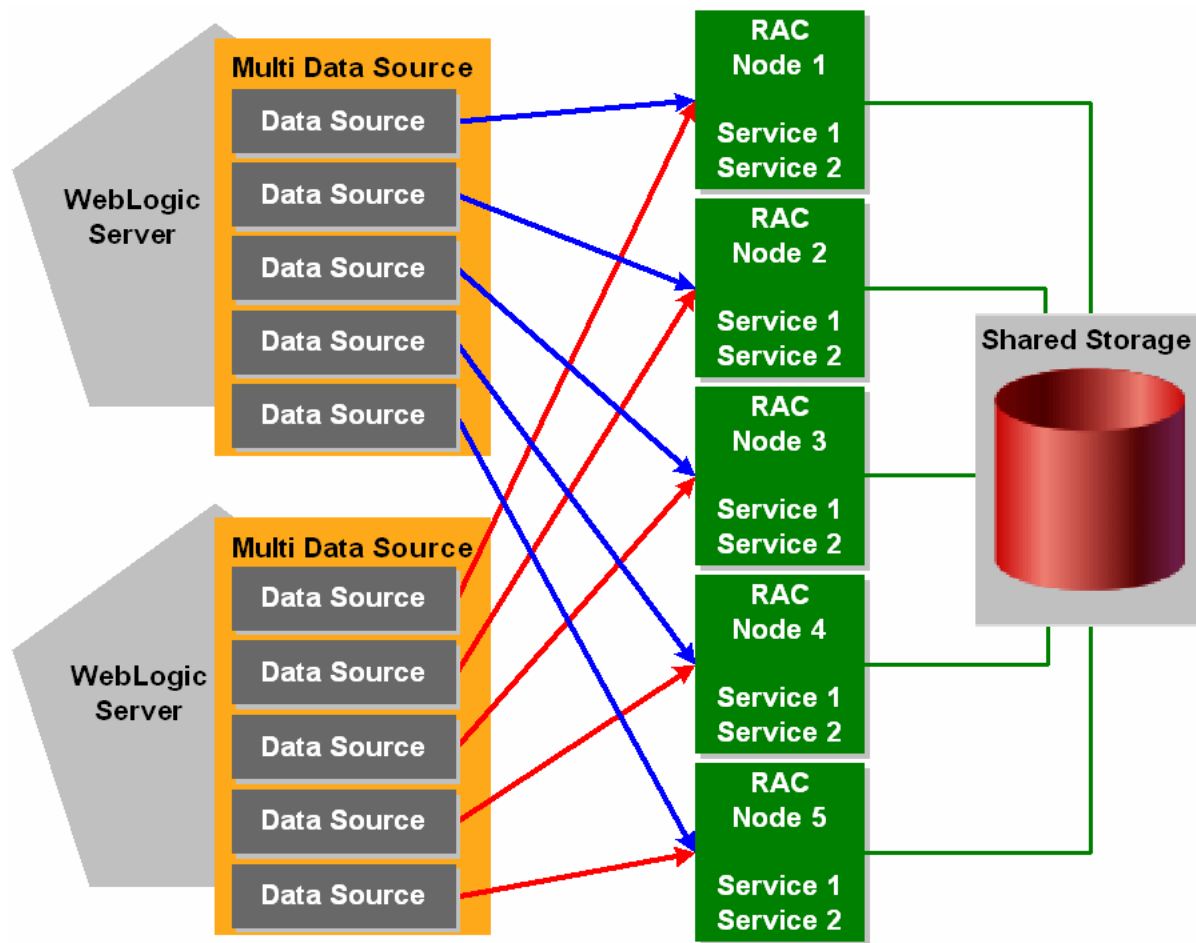
In this example, Service 1 is active on Oracle RAC Nodes 1, 2, and 3, while Service 2 is inactive on those nodes. Service 2 is active on Oracle RAC Nodes 4 and 5, while Service 1 is inactive on those nodes.

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 data source to Node 4 as needed.

C.8.2.2 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 C-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.

C.8.3 Connection Pool Capacity Planning

It is important to note the **Maximum Capacity** value you specify for a 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 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 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 data sources that can potentially create connections to a given Oracle RAC node, and the number of WebLogic server instances to which each data source/multi data source is targeted. For example, if you have one data source that is targeted to three WLS servers, that data source counts as three data sources, as each server uses its own instance of the 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 data sources. For example, assume that all available services will be actively running on the Oracle RAC node associated with each data source.

The following example explains how you could go about determining each data source's **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 your 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 10MB 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 data source is targeted to Cluster1 and is configured to connect to Service1.
- For a given Oracle RAC node, one 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 10GB of the node's memory for Service 2 and approximately 5GB for Service1.

Because Cluster1 has five WLS servers, there will be five data sources making connection requests to this Oracle RAC node. This gives you 1GB of memory available for connections from a given data source (5GB/5). Each connection requires 10MB of memory, so the **Maximum Capacity** value for each data source targeted to Cluster1 should be 100 or lower.

Because Cluster 2 has four WLS servers, there will be four data sources making connection requests to this Oracle RAC node. This gives you 2.5GB of memory available for connections from a given data source (10GB/4). Each connection requires

20MB, so the **Maximum Capacity** value for each 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 data source connecting to Service2, decrease the value for the data source connecting to Service1). As long as:

$$\begin{aligned} &(\text{Max. connections to Service1} \times \text{memory used per connection}) + \\ &(\text{Max. connections to Service2} \times \text{memory used per connection}) < \\ &\text{Available memory} \end{aligned}$$

you can avoid the potential for performance degradation or connection failures.

Alternatively, in a simple configuration, such as is shown in [Figure C-6](#), the **Maximum Capacity** value you specify for each of your data sources can be loosely determined using the following formula:

$$\text{Maximum connection pool capacity} = \frac{\text{Maximum number of connections to Oracle RAC nodes}}{(\text{Number of WebLogic Server instances} \times \text{Number of data sources targeted to each instance} \times \text{Number of active Oracle RAC services configured} \times \text{Number of Oracle RAC Nodes})}$$

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 data sources are targeted. If the data sources is targeted to a WLS cluster, this is the number of servers in the cluster.

In the example in [Figure C-6](#):

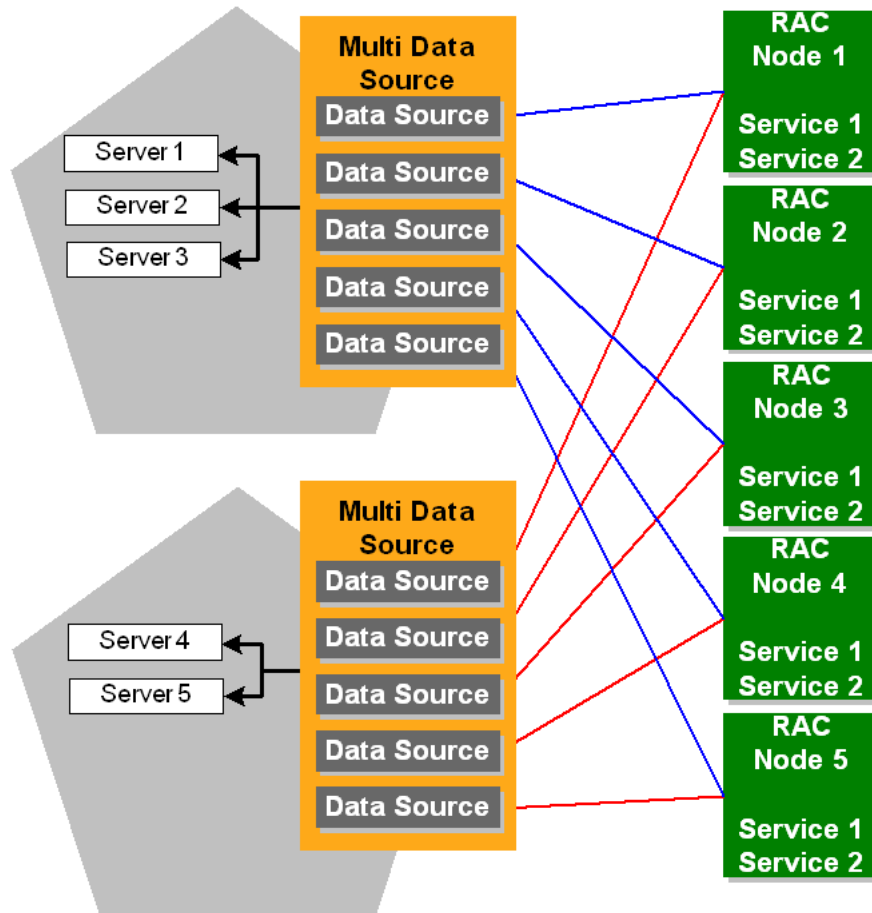
- assume that a maximum of 4000 total connections can be made to the group of Oracle RAC nodes, based on 8GB of available memory per Oracle RAC node, and 10MB of memory used per connection.
- there are a total of five server instances to which the data sources are targeted
- there are five data sources 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 data sources would be:

$$\text{Maximum connection pool capacity} = \frac{4000}{(5 \text{ server instances} \times 5 \text{ data sources} \times 2 \text{ services} \times 5 \text{ Oracle RAC nodes})}$$

which would give you a **Maximum Capacity** value of 16 for each of your data sources.

Figure C-6 Example Multi Data Source Connection Configuration



Keep in mind that this formula is just a general guideline for configuring your 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 data sources.

C.9 XA Considerations and Limitations when using multi Data Sources with Oracle RAC

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

- [Section C.9.1, "Oracle RAC XA Requirements when using multi Data Sources"](#)
- [Section C.9.2, "Known Limitations When Using Oracle RAC with multi Data Sources"](#)
- [Section C.9.3, "Known Issue Occurring After Database Server Crash"](#)

C.9.1 Oracle RAC XA Requirements when using multi Data Sources

Oracle RAC has the following requirements when using multi data sources with global transactions.

- [Section C.9.1.1, "Use Multi Data Sources"](#)
- [Section C.9.1.2, "A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the Oracle RAC Cluster"](#)
- [Section C.9.1.3, "Transaction IDs Must Be Unique Within the Oracle RAC Cluster"](#)

C.9.1.1 Use Multi Data Sources

Always use a multi data source when using XA transactions with multi data sources for Oracle RAC.

C.9.1.2 A Global Transaction Must Be Initiated, Prepared, and Concluded in the Same Instance of the Oracle RAC Cluster

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

C.9.1.3 Transaction IDs Must Be Unique Within the Oracle RAC Cluster

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.

C.9.2 Known Limitations When Using Oracle RAC with multi Data Sources

The following sections describe known issues and limitations when using XA and multi data sources with Oracle RAC:

- [Section C.9.2.1, "Potential for Data Deadlocks in Some Failure Scenarios"](#)
- [Section C.9.2.2, "Potential for Transactions Completed Out of Sequence"](#)

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

C.9.2.1 Potential for Data Deadlocks in Some Failure Scenarios

There is a window of time in which transaction IDs are not available across the Oracle RAC cluster. Because of this known limitation, after some failure conditions, some incomplete transactions cannot be properly completed, which can result in deadlocks in the database. To prevent these failure conditions from arising, WebLogic Server provides two configuration attributes that enable XA call retry for Oracle RAC: `XARetryDurationSeconds` and `XARetryIntervalSeconds`. For more information about these configuration options, see [Section C.5.3.2, "Delays During Failover."](#)

C.9.2.2 Potential for Transactions Completed Out of Sequence

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

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

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

C.9.3 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.

C.10 JDBC Store Recovery with Oracle RAC

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

- [Section C.10.1, "Configuring a JDBC Store for Use with Oracle RAC"](#)
- [Section C.10.2, "Automatic Retry"](#)

For a list of the major services that use the JDBC store, see "Monitoring Store Connections" in *Configuring Server Environments for Oracle WebLogic Server*.

C.10.1 Configuring a JDBC Store for Use with Oracle RAC

The way that a JDBC Store works limits the options you have for configuring one for use with Oracle RAC. You cannot configure a JDBC store to use a JDBC data source that is configured to support global transactions. The JDBC store must use a JDBC data source that uses a non-XA JDBC driver. For more information about this configuration option, see [Section C.7, "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.

C.10.2 Automatic Retry

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 restart WebLogic Server.

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

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

Where `x` is the number of milliseconds 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.

Using Connect-Time Failover with Oracle RAC (Deprecated)

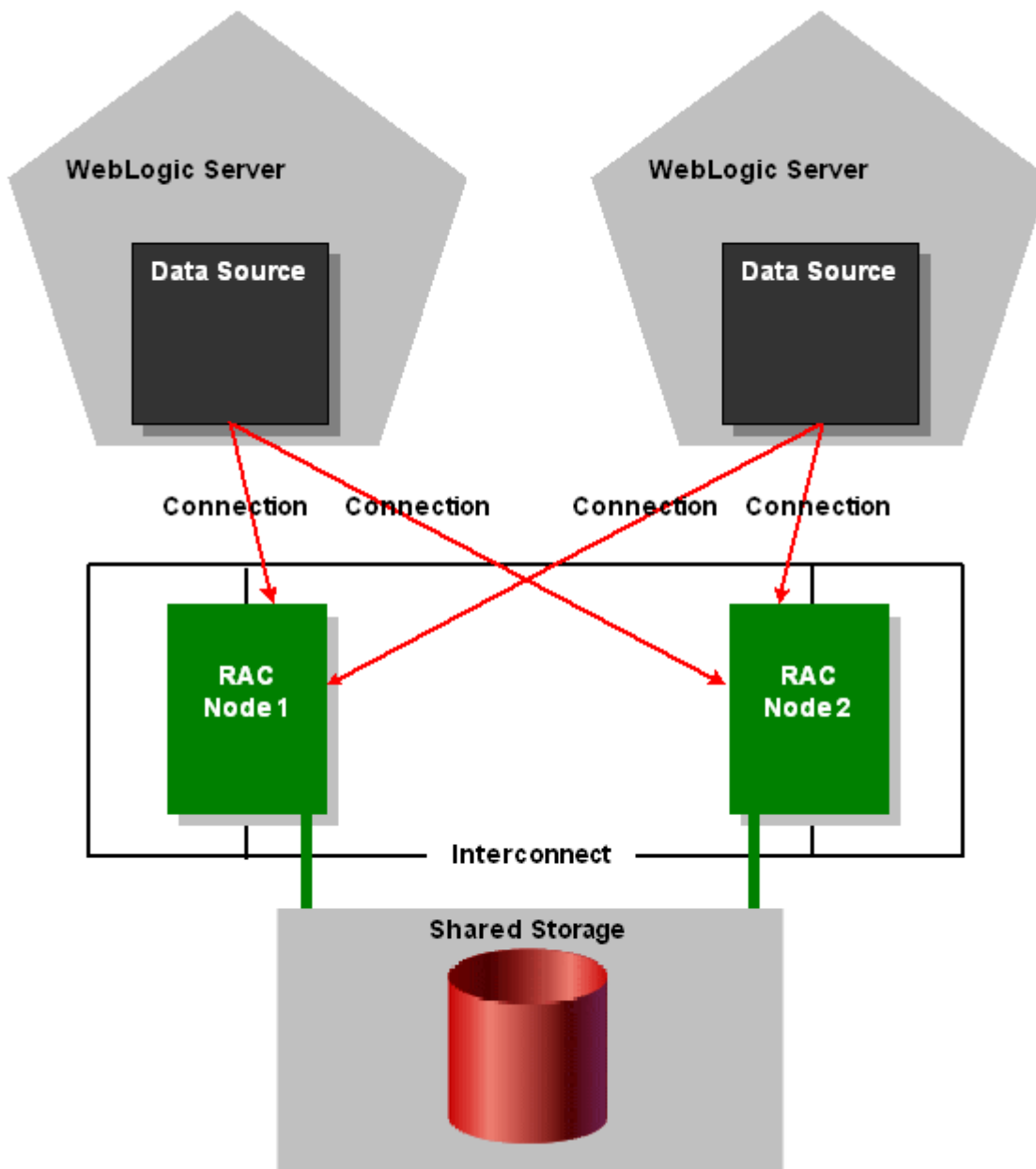
This chapter provides information on how WebLogic Server provides Connect-Time Failover (deprecated) for legacy applications that use data sources configured to use connect-time failover and load balancing.

Note: New applications should use a GridLink Data Source, which provides the same capabilities. See [Section 4.1, "What is a GridLink Data Source"](#) and "Implicit Connection Caching" in the *Database JDBC Developer's Guide*.

- [Section D.1, "Using Connect-Time Failover without Global Transactions"](#)
- [Section D.2, "Attributes of a Connect-Time Failover Configuration without Global Transactions"](#)
- [Section D.3, "Sample Configuration Code"](#)

D.1 Using Connect-Time Failover without Global Transactions

To connect WebLogic Server to multiple Oracle RAC nodes using data sources configured for connect-time failover and load balancing, configure a JDBC data source for each Oracle RAC instance in your Oracle RAC cluster with the Oracle Thin driver, as described in the sections that follow. [Figure D-1](#) shows an overview of the system.

Figure D-1 Data Source Configuration with Oracle Thin Driver Connect-Time Failover

You can use the Administration Console or any other means that you prefer to configure your domain, such as the WebLogic Scripting Tool (WLST) or a JMX program.

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

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

D.2 Attributes of a Connect-Time Failover Configuration without Global Transactions

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

- Oracle JDBC Thin driver 11g configured for connect-time failover. For example:


```
<url>jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(
  PROTOCOL=TCP)(HOST=lcqso124)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)
  (HOST=lcqso125)(PORT=1521))(FAILOVER=on)(LOAD_BALANCE=off))
  (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=snrac))</url>
<driver-name>oracle.jdbc.OracleDriver</driver-name>
```
- `ConnectionReserveTimeoutSeconds="120"`
 - Enables application requests for a connection to wait 120 seconds for a connection to become available.
- `TestConnectionsOnReserve="true"`
 - Enables testing of a database connection when an application reserves a connection from the data source. See [Section 5.4.1, "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 [Section 10.2, "Connection Testing Options for a Data Source."](#)

D.3 Sample Configuration Code

Sample configuration code is shown below.

Note: Line breaks added for readability.

```
<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>oracleRACNonXAPool</name>
  <jdbc-driver-params>
    <url>jdbc:oracle:thin:@(DESCRIPTION=
      (ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
        HOST=lcqso124)(PORT=1521))(ADDRESS=(PROTOCOL=TCP)
        (HOST=lcqso125)(PORT=1521))(FAILOVER=on)
        (LOAD_BALANCE=off))(CONNECT_DATA=(SERVER=DEDICATED)
        (SERVICE_NAME=snrac))</url>
    <driver-name>oracle.jdbc.OracleDriver</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>wlsqa</value>
      </property>
    </properties>
    <password-encrypted>{3DES}aP/xScCS8uI=</password-encrypted>
  </jdbc-driver-params>
```

```
<jdbc-connection-pool-params>
  <test-connections-on-reserve>true</test-connections-on-reserve>
  <test-table-name>SQL SELECT 1 FROM DUAL</test-table-name>
  <profile-type>4</profile-type>
</jdbc-connection-pool-params>
<jdbc-data-source-params>
  <jndi-name>oracleRACJndiName</jndi-name>
  <global-transactions-protocol>OnePhaseCommit
    </global-transactions-protocol>
</jdbc-data-source-params>
</jdbc-data-source>
```

Using Fast Connection Failover with Oracle RAC

This chapter provides information on how to use WebLogic server with Oracle Fast Connection Failover.

WebLogic Server supports Fast Connection Failover, an Oracle feature which 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.

For more information, see "Fast Connection Failover" at http://download-east.oracle.com/docs/cd/B19306_01/java.102/b14355/fstconfo.htm#CIHJBFFC in the *Oracle® Database JDBC Developer's Guide and Reference*.

E.1 JDBC Driver Configuration for use with Oracle Fast Connection Failover

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

- In Driver Class Name—set the class name to `oracle.jdbc.pool.OracleDataSource`.
- In Properties—set the ONS configuration string to remotely subscribe the 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.

Because the Oracle driver uses ONS to implement Fast Connection Failover, you must add `ons.jar` to the WebLogic classpath. To do so, edit the `setDomainEnv` script in the domain's `bin` directory, and add `ons.jar` to the `POST_CLASSPATH` variable.

