Oracle® Fusion Middleware

Developing JTA Applications for Oracle WebLogic Server



26 (26.1.0.0.0)
G56956-01
November 2026

ORACLE

Oracle Fusion Middleware Developing JTA Applications for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56956-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

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

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

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

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

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

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

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

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

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.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation i
Conventions ii

1 Introducing Transactions

Overview of Transactions in WebLogic Server Applications 1
- ACID Properties of Transactions 1
- Supported Programming Model 2
- Supported API Models 2
- Distributed Transactions and the Two-Phase Commit Protocol 2
- Support for Business Transactions 3

When to Use Transactions 3
What Happens During a Transaction 4
- Transactions in WebLogic Server EJB Applications 4
  - Container-managed Transactions 5
  - Bean-managed Transactions 6
- Transactions in WebLogic Server RMI Applications 6

Transactions Sample Code 8
- Transactions Sample EJB Code 8
  - Importing Packages 8
  - Using JNDI to Return an Object Reference 9
  - Starting a Transaction 9
  - Completing a Transaction 9
- Transactions Sample RMI Code 10
  - Importing Packages 10
  - Using JNDI to Return an Object Reference to the UserTransaction Object 10
  - Starting a Transaction 11
  - Completing a Transaction 11

Copyright © 2007, 2026, Oracle and/or its affiliates.

2 Configuring Transactions

Overview of Transaction Configuration 1
Configuring JTA 2
- Unregister Resource Grace Period 6
- Additional Attributes for Managing Transactions 6
- XA Transaction Cluster Affinity 7

Configuring Network Channels for JTA Communication 7
Using Transaction Log Files to Recover Transactions 11
- Using the Default Persistent Store 11
  - Setting the Path for the Default Persistent Store 11
  - Setting the Default Persistent Store Synchronous Write Policy 11
- Using a JDBC TLOG Store 11
- Last Logging Resource 12
- XA Transactions without Transaction TLog Write 12
  - What is a Determiner Resource? 12
  - Configuring XA Transactions without TLogs 12

Read-Only, One-Phase Commit Optimizations 15
- Configuring Read-only, One-phase Commit Optimization and Two-phase Commit Disablement 16
- Monitoring Read-only, One-phase Transaction Statistics 16

3 Managing Transactions

Monitoring Transactions 1
Handling Heuristic Completions 1
Moving a Server 2
Abandoning Transactions 3
- Transaction Completion Timeout 3

Manually Resolving Current (Inflight) Transactions 4
- Manual Commit and Rollback Options 5

Transaction Recovery After a Server Fails 6
- Transaction Recovery Service Actions After a Crash 7
- Recovering Transactions For a Failed Non-Clustered Server 8
- Recovering Transactions For a Failed Clustered Server 9
  - Server Migration 9
  - Automatic Transaction Recovery Service Migration 9
  - Manual Transaction Recovery Service Migration 10
  - Managed Server Independence 10
  - Limitations of Migrating the Transaction Recovery Service 11
  - Preparing to Migrate the Transaction Recovery Service 11
  - Constraining Servers to Which the Transaction Recovery Service Can Migrate 12
  - Manually Migrating the Transaction Recovery Service to the Original Server 13

Copyright © 2007, 2026, Oracle and/or its affiliates.

How to Remove Transaction Records 13
How to Remove the TLog in the LLR Database 14
How to Remove the TLog Files from the Default Store 14
How to Remove the TLog from a JDBC TLog Store 14

4 Using Transaction Guard

Overview of Transaction Guard 1
Enabling Transaction Guard 2
Enabling Transaction Guard for WebLogic Data Sources 2
Enabling Transaction Guard for Database Service 3
Data Source Statistics for Transaction Guard 3

5 Transaction Service

About the Transaction Service 1
Capabilities and Limitations 2
Lightweight Clients with Delegated Commit 2
Client-initiated Transactions 2
Transaction Integrity 2
Transaction Termination 3
Flat Transactions 3
Relationship of the Transaction Service to Transaction Processing 3
Multithreaded Transaction Client Support 3
Transaction Id 3
Transaction Name and Properties 4
Transaction Status 4
Transaction Statistics 4
General Constraints 4
Transaction Scope 5
Transaction Service in EJB Applications 5
Transaction Service in RMI Applications 5
Transaction Service Interoperating with OTS 5
Server-Server 2PC 5
Client Demarcated Transactions 6

6 Transaction Recovery Spanning Multiple Sites or Data Centers

Understanding XA Transaction Recovery in Disaster Recovery 1
Active-Passive XA Transaction Recovery 2
Example Active-Passive Domain Configuration for XA Transaction Recovery 3
Active-Active Stretch Cluster XA Transaction Recovery 3

Copyright © 2007, 2026, Oracle and/or its affiliates.

Example Active-Active Stretch Cluster for XA Transaction Recovery 4
Additional Information on Maximum Availability Architecture 5

7 Configuring Secure Inter-Domain and Intra-Domain Transaction Communication

What is Secure Inter-Domain and Intra-Domain Transaction Communication? 1
Requirements for Transaction Communication 1
How to Determine the Communication to Use for Domain Transactions 2
Configuring Secure Channel Communication 3
Local Domain Security 3
Cross Domain Security 4
Important Considerations When Configuring Cross Domain Security 4
Cross Domain Security is Not Transitive 5
Adding Domains to the Exclude List Based on Transaction Participation 6

8 Java Transaction API and Oracle WebLogic Extensions

JTA API Overview 1
Oracle WebLogic Extensions to JTA 2

9 Logging Last Resource Transaction Optimization

About the LLR Optimization Transaction Optimization 2
Logging Last Resource Processing Details 2
LLR Database Table Details 3
LLR Table Transaction Log Records 4
Failure and Recovery Processing for LLR 4
Coordinating Server Crash 4
JDBC Connection Failure 5
LLR Transaction Recover During Server Startup 5
Failover Considerations for LLR 5
Optimizing Performance with LLR 5
Optimizing Transaction Coordinator Location 6
Varied Performance for Read-Only Operations Through an LLR Data Source 6
Dedicating LLR Tables by Data Source 6
Limitations 7
First Resource Commit Ordering 7

10 Transactions in EJB Applications

Before You Begin 1

Copyright © 2007, 2026, Oracle and/or its affiliates.

General Guidelines 2
Transaction Attributes 2
About Transaction Attributes for EJBs 2
Transaction Attributes for Container-Managed Transactions 3
Transaction Attributes for Bean-Managed Transactions 3
Participating in a Transaction 4
Transaction Semantics 4
Transaction Semantics for Container-Managed Transactions 4
Transaction Semantics for Stateful Session Beans 4
Transaction Semantics for Stateless Session Beans 5
Transaction Semantics for Entity Beans 5
Transaction Semantics for Bean-Managed Transactions 6
Transaction Semantics for Stateful Session Beans 6
Transaction Semantics for Stateless Session Beans 7
Session Synchronization 7
Synchronization During Transactions 7
Setting Transaction Timeouts 8
Handling Exceptions in EJB Transactions 8

11 Transactions in RMI Applications

Before You Begin 1
General Guidelines 1

12 Using JDBC XA Drivers with WebLogic Server

Using Oracle Thin/XA Driver 1
Set the Environment for the Oracle Thin/XA Driver 1
Configure WebLogic Server 1
Enable XA on the Database Server 2
Oracle Thin/XA Driver Configuration Properties 2
Using Other XA Drivers 2
Using WebLogic-branded Data Direct Drivers 2
Additional Considerations 2

13 Coordinating XAResources with the WebLogic Server Transaction Manager

Overview of Coordinating Distributed Transactions with Foreign XAResources 2
Registering an XAResource to Participate in Transactions 3
Enlisting and Delisting an XAResource in a Transaction 5
Standard Enlistment 6

Copyright © 2007, 2026, Oracle and/or its affiliates.

Dynamic Enlistment 7
Static Enlistment 7
Commit processing 8
Recovery 8
Resource Health Monitoring 9
Jakarta EE Connector Architecture Resource Adapter 10
Implementation Tips 10
Sharing the WebLogic Server Transaction Log 10
Transaction global properties 11
TxHelper.createXid 11
Changes in the Resource Registration Name 11
FAQs 12
Additional Documentation about JTA 12

14 Participating in Transactions Managed by a Third-Party Transaction Manager

Overview of Participating in Foreign-Managed Transactions 1
Importing Transactions with the Client Interposed Transaction Manager 2
Get the Client Interposed Transaction Manager 4
Get the XAResource from the Interposed Transaction Manager 4
Cluster-wide Recovery 4
Limitations of the Client Interposed Transaction Manager 5
Importing Transactions with the Server Interposed Transaction Manager 5
Get the Server Interposed Transaction Manager 6
Limitations of the Server Interposed Transaction Manager 6
Transaction Processing for Imported Transactions 6
Transaction Processing Limitations for Imported Transactions 7
Commit Processing for Imported Transactions 7
Recovery for Imported Transactions 8
Transactions that Span Transaction Manager Systems 8

15 Troubleshooting Transactions

Overview 1
Troubleshooting Tools 1
Exceptions 1
Transaction Identifier 2
Transaction Name and Properties 2
Transaction Status 2
Transaction Statistics 2
Transaction Monitoring 3

Copyright © 2007, 2026, Oracle and/or its affiliates.

Debugging JTA Resources 3
Enabling Debugging 3
Enable Debugging Using the Command Line 3
Enable Debugging Using the WebLogic Remote Console 3
Enable Debugging Using the WebLogic Scripting Tool 4
Changes to the config.xml File 5
JTA Debugging Scopes 5

Index

Copyright © 2007, 2026, Oracle and/or its affiliates.

Preface

This document explains developing JTA applications for Oracle WebLogic Server.

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

Audience

This document is written for application developers who are interested in building transactional Java applications that run in the WebLogic Server environment. It is assumed that readers are familiar with the WebLogic Server platform, Java Platform, Enterprise Edition (Jakarta EE) programming, and transaction processing concepts.

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 customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

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

Related Documentation

Samples and Tutorials

Oracle provides a variety of code examples and tutorials that show WebLogic Server configuration and API use, and provide practical instructions on how to perform key

Copyright © 2007, 2026, Oracle and/or its affiliates.

development tasks. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

New and Changed WebLogic Server Features

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

Conventions

The following text conventions are used in this document:

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

1

Introducing Transactions

Understand the basics of transaction such as their ACID properties, supported programming and API models, distributed transactions, and so on. Also, learn when to use a transaction and how a transaction is processed with help of a example transaction code.

- Overview of Transactions in WebLogic Server Applications
  Learn about the ACID properties of transactions, supported programming and API models, distributed transactions, two-phase protocol and support for business transaction.
- When to Use Transactions
  Learn about the situations in which you can use transactions supported by WebLogic Server.
- What Happens During a Transaction
  Learn the difference in how transactions work in WebLogic Server EJB applications and WebLogic Server RMI applications.
- Transactions Sample Code
  Learn about transaction with the help of EJB and RMI sample codes.

Overview of Transactions in WebLogic Server Applications

Learn about the ACID properties of transactions, supported programming and API models, distributed transactions, two-phase protocol and support for business transaction.

- ACID Properties of Transactions
- Supported Programming Model
- Supported API Models
- Distributed Transactions and the Two-Phase Commit Protocol
- Support for Business Transactions

ACID Properties of Transactions

A fundamental feature of WebLogic Server is transaction management. Transactions are a means to guarantee that database changes are completed accurately and that they take on all the **ACID properties** of a high-performance transaction, including:

- Atomicity—all changes that a transaction makes to a database are made as one unit; otherwise, all changes are rolled back.
- Consistency—a successful transaction transforms a database from a previous valid state to a new valid state.
- Isolation—changes that a transaction makes to a database are not visible to other operations until the transaction completes its work.
- Durability—changes that a transaction makes to a database survive future system or media failures.

Copyright © 2007, 2026, Oracle and/or its affiliates.

WebLogic Server protects the integrity of your transactions by providing a complete infrastructure for ensuring that database updates are done accurately, even across a variety of resource managers. If any one operation fails, the entire set of operations is rolled back.

Supported Programming Model

WebLogic Server supports transactions in the Java Platform, Enterprise Edition (Jakarta EE) programming model. WebLogic Server provides full support for transactions in Java applications that use Jakarta Enterprise Beans, in compliance with the Jakarta Enterprise Beans (EJBs) Specification 4.0. WebLogic Server also supports the Java Transaction API (JTA) Specification 2.0. Both specifications are published at the following locations:

Specification	Location
EJB 4.0	https://jakarta.ee/specifications/enterprise-beans/4.0/
JTA 2.0	https://jakarta.ee/specifications/transactions/2.0/

Supported API Models

WebLogic Server supports the Java Transaction API (JTA), which is used by:

- Enterprise JavaBean (EJB) applications within the WebLogic Server EJB container.
- Remote Method Invocation (RMI) applications within the WebLogic Server infrastructure.

For information, see the following API Javadoc.

- https://jakarta.ee/specifications/transactions/2.0/apidocs/
- https://docs.oracle.com/en/java/javase/17/docs/api/java.transaction.xa/module-summary.html

Distributed Transactions and the Two-Phase Commit Protocol

WebLogic Server supports distributed transactions and the two-phase commit protocol for enterprise applications. A **distributed transaction** is a transaction that updates multiple resource managers (such as databases) in a coordinated manner. In contrast, a **local transaction** begins and commits the transaction to a single resource manager that internally coordinates API calls; there is no transaction manager. The **two-phase commit protocol** is a method of coordinating a single transaction across two or more resource managers. It guarantees data integrity by ensuring that transactional updates are committed in all of the participating databases, or are fully rolled back out of all the databases, reverting to the state prior to the start of the transaction. In other words, either all the participating databases are updated, or none are updated.

Distributed transactions involve the following participants:

- Transaction originator—initiates the transaction. The transaction originator can be a user application, an Enterprise JavaBean, or a JMS client.
- Transaction manager—manages transactions on behalf of application programs. A transaction manager coordinates commands from application programs to start and complete transactions by communicating with all resource managers that are participating in those transactions. When resource managers fail during transactions, transaction managers help resource managers decide whether to commit or roll back pending transactions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Recoverable resource—provides persistent storage for data. The resource is most often a database.
- Resource manager—provides access to a collection of information and processes. Transaction-aware JDBC drivers are common resource managers. Resource managers provide transaction capabilities and permanence of actions; they are entities accessed and controlled within a distributed transaction. The communication between a resource manager and a specific resource is called a **transaction branch**.

The first phase of the two-phase commit protocol is called the **prepare phase**. The required updates are recorded in a transaction log file, and the resource must indicate, through a resource manager, that it is ready to make the changes. Resources either vote to commit the updates or to roll back to the previous state. What happens in the second phase depends on how the resources vote. If all resources vote to commit, all the resources participating in the transaction are updated. If one or more of the resources vote to roll back, then all the resources participating in the transaction are rolled back to their previous state.

Support for Business Transactions

WebLogic JTA provides the following support for your business transactions:

- Creates a unique transaction identifier when a client application initiates a transaction.
- Supports an optional transaction name describing the business process that the transaction represents. The transaction name makes statistics and error messages more meaningful.
- Works with the WebLogic Server infrastructure to track objects that are involved in a transaction and, therefore, coordinates these objects when the transaction is ready to commit.
- Notifies the resource managers—which are, most often, databases—when they are accessed on behalf of a transaction. Resource managers then lock the accessed records until the end of the transaction.
- Orchestrates the two-phase commit when the transaction completes, which ensures that all the participants in the transaction commit their updates simultaneously. It coordinates the commit with any databases that are being updated using Open Group's XA protocol. Many popular relational databases support this standard.
- Executes the rollback procedure when the transaction must be stopped.
- Executes a recovery procedure when failures occur. It determines which transactions were active in the machine at the time of the crash, and then determines whether the transaction should be rolled back or committed.
- Manages transaction timeouts. If a business operation takes too much time or is only partially completed due to failures, the system takes action to automatically issue a timeout for the transaction and free resources, such as database locks.

When to Use Transactions

Learn about the situations in which you can use transactions supported by WebLogic Server.

Transactions are appropriate in the situations described in the following list. Each situation describes a transaction model supported by the WebLogic Server system. Keep in mind that distributed transactions should not span more than a single user input screen; more complex, higher level transactions are best implemented with a series of distributed transactions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Within the scope of a single client invocation on an object, the object performs multiple edits to data in a database. If one edits fails, the object needs a mechanism to roll back all the edits. (In this situation, the individual database edits are not necessarily EJB or RMI invocations. A client, such as an applet, obtain a reference to the `Transaction` and `TransactionManager` objects, using JNDI, and start a transaction.)

  For example, consider a banking application. The client invokes the transfer operation on a teller object. The transfer operation requires the teller object to make the following invocations on the bank database:

  – Invoking the debit method on one account.

  – Invoking the credit method on another account.

  If the credit invocation on the bank database fails, the banking application needs a way to roll back the previous debit invocation.

- The client application needs a conversation with an object managed by the server application, and the client application makes multiple invocations on a specific object instance. The conversation may be characterized by one or more of the following:

  – Data is cached in memory or written to a database during or after each successive invocation.

  – Data is written to a database at the end of the conversation.

  – The client application needs the object to maintain an in-memory context between each invocation; that is, each successive invocation uses the data that is being maintained in memory across the conversation.

  – At the end of the conversation, the client application needs the ability to cancel all database write operations that may have occurred during or at the end of the conversation.

What Happens During a Transaction

Learn the difference in how transactions work in WebLogic Server EJB applications and WebLogic Server RMI applications.

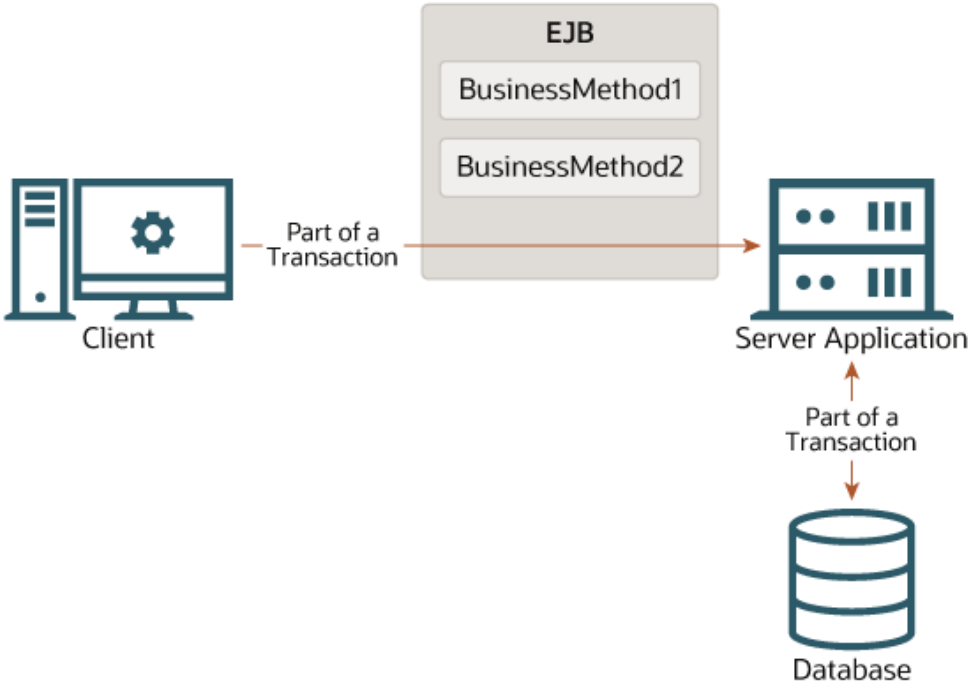
- Transactions in WebLogic Server EJB Applications
- Transactions in WebLogic Server RMI Applications

Transactions in WebLogic Server EJB Applications

Figure 1-1 illustrates how transactions work in a WebLogic Server EJB application.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Figure 1-1 How Transactions Work in a WebLogic Server EJB Application



WebLogic Server supports two types of transactions in WebLogic Server EJB applications:

- In **container-managed transactions**, the WebLogic Server EJB container manages the transaction demarcation. Transaction attributes in the EJB deployment descriptor determine how the WebLogic Server EJB container handles transactions with each method invocation. For more information about the deployment descriptor, see Implementing Enterprise Java Beans in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.
- In **bean-managed transactions**, the EJB manages the transaction demarcation. The EJB makes explicit method invocations on the `UserTransaction` object to begin, commit, and roll back transactions. See `weblogic.transaction.UserTransaction` in the *Java API Reference for Oracle WebLogic Server*.

The sequence of transaction events differs between container-managed and bean-managed transactions.

- Container-managed Transactions
- Bean-managed Transactions

Container-managed Transactions

For EJB applications with container-managed transactions, a basic transaction works in the following way:

1. In the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the transaction type (`transaction-type` element) for container-managed demarcation (`Container`).
2. In the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the default transaction attribute (`trans-attribute` element) for the EJB, which is one of the following settings: `NotSupported`, `Required`, `Supports`, `RequiresNew`, `Mandatory`, or

Copyright © 2007, 2026, Oracle and/or its affiliates.

`Never`. For a detailed description of these settings, see Section 17.6.2 in the Jakarta Enterprise Beans Specification 2.0.

3. Optionally, in the EJB's deployment descriptor, the Bean Provider or Application Assembler specifies the `trans-attribute` for one or more methods.
4. When a client application invokes a method in the EJB, the EJB container checks the `trans-attribute` setting in the deployment descriptor for that method. If no setting is specified for the method, the EJB uses the default `trans-attribute` setting for that EJB.
5. The EJB container takes the appropriate action depending on the applicable `trans-attribute` setting.
   - For example, if the `trans-attribute` setting is `Required`, the EJB container invokes the method within the existing transaction context or, if the client called without a transaction context, the EJB container begins a new transaction before executing the method.
   - For `Mandatory`, the method executes in the caller's transaction context. Without a caller transaction, a business-view invocation throws `jakarta.ejb.EJBTransactionRequiredException`; the Enterprise Beans 2.1 remote view throws `jakarta.transaction.TransactionRequiredException`, and its local view throws `jakarta.ejb.TransactionRequiredLocalException`.
6. During invocation of the business method, if it is determined that a rollback is required, the business method calls the `EJBContext.setRollbackOnly` method, which notifies the EJB container that the transaction is to be rolled back at the end of the method invocation.

   > **Note**
   >
   > Calling the `EJBContext.setRollbackOnly` method is allowed only for methods that have a meaningful transaction context.

7. At method return, the container completes a transaction it started. A transaction supplied by the caller remains under the caller's control.

Bean-managed Transactions

For EJB applications with bean-managed transaction demarcations, a basic transaction works in the following way:

1. Set the bean transaction management type to `Bean`.
2. The bean obtains a `UserTransaction` through `EJBContext.getUserTransaction()` or `java:comp/UserTransaction`.
3. The bean calls `begin()` before its transactional work.
4. The bean calls `commit()` on success or `rollback()` when its work fails.
5. The transaction manager coordinates the enlisted resource managers.

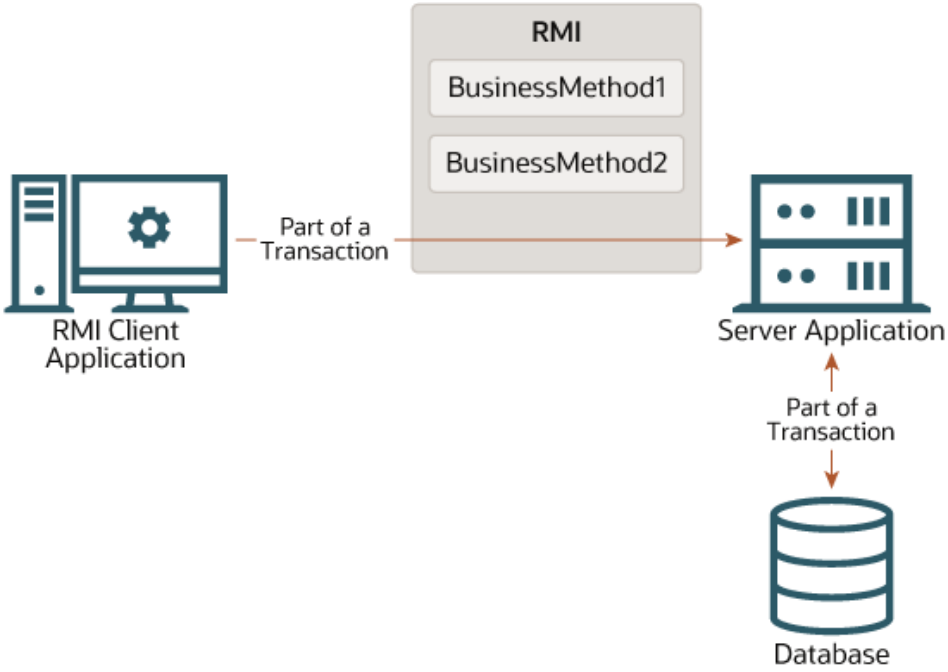
The container suspends any client transaction before invoking a bean-managed business method. A stateful session bean can retain its own transaction between calls; stateless and singleton session beans must complete it before returning.

Transactions in WebLogic Server RMI Applications

Figure 1-2 illustrates how transactions work in a WebLogic Server RMI application.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Figure 1-2 How Transactions Work in a WebLogic Server RMI Application



For RMI client and server applications, a basic transaction works in the following way:

1. The application uses JNDI to return an object reference to the `UserTransaction` object for the WebLogic Server domain.

   Obtaining the object reference begins a conversational state between the application and that object. The conversational state continues until the transaction is completed (committed or rolled back). Once instantiated, RMI objects remain active in memory until they are released (typically during server shutdown). For the duration of the transaction, the WebLogic Server infrastructure does not perform any deactivation or activation.
2. The client application begins a transaction using the `UserTransaction.begin` method, and issues a request to the server application. All operations on the server application execute within the scope of a transaction.
   - If a call to any of these operations raises an exception (either explicitly or because of a communication failure), catch the exception and the use the `UserTransaction.rollback` method to roll back the transaction.
   - If no exceptions occur, the client application commits the current transaction using the `UserTransaction.commit` method. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.
3. The `UserTransaction.commit` method causes WebLogic Server to call the transaction manager to complete the transaction.
4. The transaction manager is responsible for coordinating with the resource managers to update any databases.

See Transactions in RMI Applications.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Transactions Sample Code

Learn about transaction with the help of EJB and RMI sample codes.

- Transactions Sample EJB Code
- Transactions Sample RMI Code

Transactions Sample EJB Code

This section provides a walkthrough of sample code fragments from a class in an EJB application. This topic includes the following sections:

The code fragments demonstrate using the `UserTransaction` object for *bean-managed* transaction demarcation. The deployment descriptor for this bean specifies the transaction type (`transaction-type` element) for transaction demarcation (`Bean`).

> **Note**
>
> In a global transaction, use a database connection from a local JDBC data source—on the WebLogic Server instance on which the EJB is running. Do not use a connection from a JDBC data source on a remote WebLogic Server instance.
>
> These code fragments do not derive from any of the sample applications that ship with WebLogic Server. They simply illustrate the use of the `UserTransaction` object within an EJB application.

- Importing Packages
- Using JNDI to Return an Object Reference
- Starting a Transaction
- Completing a Transaction

Importing Packages

Example 1-1 shows importing the necessary packages for transactions, including:

- `jakarta.transaction.UserTransaction`. For a list of methods associated with this object, see the online Javadoc.
- System exceptions. For a list of exceptions, see the online Javadoc.

Example 1-1 Importing Packages

```
import javax.naming.*;
import jakarta.transaction.UserTransaction;
import jakarta.transaction.SystemException;
import jakarta.transaction.HeuristicMixedException;
import jakarta.transaction.HeuristicRollbackException;
import jakarta.transaction.NotSupportedException;
import jakarta.transaction.RollbackException;
import java.sql.*;
import java.util.*;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Using JNDI to Return an Object Reference

Example 1-2 shows how look up an object on the JNDI tree.

Example 1-2 Performing a JNDI Lookup

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

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

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

ctx = new InitialContext(env);

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

Starting a Transaction

Example 1-3 shows starting a transaction by getting a `UserTransaction` object and calling the `jakarta.transaction.UserTransaction.begin()` method. Database operations that occur after this method invocation and prior to completing the transaction exist within the scope of this transaction.

Example 1-3 Starting a Transaction

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

Completing a Transaction

Example 1-4 shows completing the transaction depending on whether an exception was thrown during any of the database operations that were attempted within the scope of this transaction:

- If an exception was thrown during any of the database operations, the application calls the `jakarta.transaction.UserTransaction.rollback()` method.
- If no exception was thrown, the application calls the `jakarta.transaction.UserTransaction.commit()` method to attempt to commit the transaction after all database operations completed successfully. Calling this method ends the transaction and starts the processing of the operation, causing the WebLogic Server EJB container to call the transaction manager to complete the transaction. The transaction is committed only if all of the participants in the transaction agree to commit.

Example 1-4 Completing a Transaction

```
tx.commit();

// or:

tx.rollback();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Transactions Sample RMI Code

This topic provides a walkthrough of sample code fragments from a class in an RMI application. This topic includes the following sections:

The code fragments demonstrate using the `UserTransaction` object for RMI transactions. For guidelines on using transactions in RMI applications, see Transactions in RMI Applications.

> **Note**
>
> These code fragments do not derive from any of the sample applications that ship with WebLogic Server. They simply illustrate the use of the `UserTransaction` object within an RMI application.

- Importing Packages
- Using JNDI to Return an Object Reference to the UserTransaction Object
- Starting a Transaction
- Completing a Transaction

Importing Packages

Example 1-5 shows importing the necessary packages, including the following packages used to handle transactions:

- `jakarta.transaction.UserTransaction`. For a list of methods associated with this object, see the online Javadoc.
- System exceptions. For a list of exceptions, see the online Javadoc.

Example 1-5 Importing Packages

```
import javax.naming.*;
import java.rmi.*;
import jakarta.transaction.UserTransaction;
import jakarta.transaction.SystemException;
import jakarta.transaction.HeuristicMixedException;
import jakarta.transaction.HeuristicRollbackException;
import jakarta.transaction.NotSupportedException;
import jakarta.transaction.RollbackException;
import java.sql.*;
import java.util.*;
```

After importing these classes, initialize an instance of the `UserTransaction` object to null.

Using JNDI to Return an Object Reference to the UserTransaction Object

Example 1-6 shows searching the JNDI tree to return an object reference to the `UserTransaction` object for the appropriate WebLogic Server domain.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Obtaining the object reference begins a conversational state between the application and that object. The conversational state continues until the transaction is completed (committed or rolled back). Once instantiated, RMI objects remain active in memory until they are released (typically during server shutdown). For the duration of the transaction, the WebLogic Server infrastructure does not perform any deactivation or activation.

Example 1-6 Performing a JNDI Lookup

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

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

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

ctx = new InitialContext(env);

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

Starting a Transaction

Example 1-7 shows starting a transaction by calling the `jakarta.transaction.UserTransaction.begin()` method. Database operations that occur after this method invocation and prior to completing the transaction exist within the scope of this transaction.

Example 1-7 Starting a Transaction

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

Completing a Transaction

Example 1-8 shows completing the transaction depending on whether an exception was thrown during any of the database operations that were attempted within the scope of this transaction:

- If an exception was thrown, the application calls the `jakarta.transaction.UserTransaction.rollback()` method if an exception was thrown during any of the database operations.
- If no exception was thrown, the application calls the `jakarta.transaction.UserTransaction.commit() method` to attempt to commit the transaction after all database operations completed successfully. Calling this method ends the transaction and starts the processing of the operation, causing WebLogic Server to call

Copyright © 2007, 2026, Oracle and/or its affiliates.

the transaction manager to complete the transaction. The transaction is committed only if all of the participants in the transaction agree to commit.

Example 1-8 Completing a Transaction

```
tx.commit();

// or:

tx.rollback();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

2

Configuring Transactions

Learn basic configuration tasks related to transactions. These tasks include using JTA, configuring secure transaction communication, using transaction log (TLog) files, and using read-only, one-phase commit optimizations.
This chapter includes the following sections:

- Overview of Transaction Configuration
  Transaction Configuration can be handled through the Remote Console that provides the interface for configuring features of WebLogic Server, including WebLogic JTA.
- Configuring JTA
  You can configure JTA settings at the domain or cluster level. Once you configure WebLogic JTA and any transaction participants, the system manages transactions using the JTA API and the WebLogic JTA extensions.
- Configuring Network Channels for JTA Communication
  Network channels are a configurable resource in WebLogic Server that you can configure to separate and load balance network traffic. Also, you can configure network channels to separate internal server communication from client user communication.
- Using Transaction Log Files to Recover Transactions
  Each server has a transaction log which stores information about committed transactions coordinated by the server that may not have been completed. WebLogic Server uses the transaction log (TLogs) when recovering from system crashes or network failures.
- Read-Only, One-Phase Commit Optimizations

Overview of Transaction Configuration

Transaction Configuration can be handled through the Remote Console that provides the interface for configuring features of WebLogic Server, including WebLogic JTA.

The configuration process involves specifying values for attributes. These attributes define the transaction environment, including the following:

- Transaction timeouts and limits
- Transaction manager behavior

You should also be familiar with the administration of Jakarta EE components that participate in transactions, such as EJBs, JDBC data sources, and JMS.

> **Note**
>
> You can also use the WebLogic Scripting Tool (WLST; see *Understanding the WebLogic Scripting Tool*) or JMX (see *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*) to configure transaction-related settings.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Configuring JTA

You can configure JTA settings at the domain or cluster level. Once you configure WebLogic JTA and any transaction participants, the system manages transactions using the JTA API and the WebLogic JTA extensions.

Note the following:

- Configuration settings for JTA (transactions) are applicable at the domain and cluster level:
  - At the domain level, attribute settings apply to all servers within a domain. These settings are superseded by any settings at the cluster level. See Table 2-1
  - At the cluster level, attribute settings apply to a cluster within a domain. These settings supersede any settings at the domain level.
- Monitoring tasks for JTA are performed at the server level.
- Configuration settings for participating resources (such as JDBC data sources) are per configured object. The settings apply to all instances of a particular object. See JDBC Data Source Transaction Options in *Administering JDBC Data Sources for Oracle WebLogic Server* and Configure Global Transaction Options for a JDBC Data Source in the *Oracle WebLogic Remote Console Online Help*.

Table 2-1 Configuration Options for JTA (transactions)

Name	Description
Timeout Seconds	Specifies the maximum amount of time, in seconds, an active transaction is allowed to be in the first phase of a two-phase commit transaction. If the specified amount of time expires, the transaction is automatically rolled back. MBean Attribute: JTAMBean.TimeoutSeconds Minimum value: 1 Maximum value: 2147483647 **Note:** The DBMS has its own `DISTRIBUTED_LOCK_TIMEOUT`, with a default of 60 seconds, the maximum period it will allow an XA transaction to hold DBMS locks. It is important that this DBMS option be tuned to accommodate the longest expected WebLogic transaction. If this DBMS timeout is lower than the WebLogic JTA/tx timeout, the DBMS may roll back and 'forget' an on-going transaction out from under WebLogic. This will cause 'NOTA' (transaction ID unknown) errors in WebLogic when WebLogic subsequently asks the DBMS to do something more on behalf of the transaction, and the DBMS now knows nothing about it.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Configuration Options for JTA (transactions)

Name	Description
Abandon Timeout Seconds	Specifies the maximum amount of time, in seconds, a transaction manager persists in attempting to complete the second phase of a two-phase commit transaction. During the second phase of a two-phase commit transaction, the transaction manager continues to try to complete the transaction until all resource managers indicate that the transaction is completed. After the abandon transaction timer expires, no further attempt is made to resolve the transaction. If the transaction is in a prepared state before being abandoned, the transaction manager rolls back the transaction to release any locks held on behalf of the abandoned transaction. MBean Attribute: JTAMBean.AbandonTimeoutSeconds Minimum value: 1 Maximum value: 2147483647 **Note:** The `Abandon Timeout Seconds` should not be smaller than the `Timeout Seconds`, as the transactions that are still active can be abandoned and remain pending. To avoid this, when the `Abandon Timeout Seconds` is configured too low, it will be auto-adjusted to equal the `Timeout Seconds`.
Before Completion Iteration Limit	The maximum number of cycles that the transaction manager performs the beforeCompletion synchronization callback for this WebLogic Server domain. Nothing prevents a Synchronization object from registering another during beforeCompletion, even those whose beforeCompletions have already been called. For example, an EJB can call another in its `ejbStore()` method. To accommodate this, the transaction manager calls all Synchronization objects, then repeats the cycle if new ones have been registered. This count sets a limit to the number of cycles that synchronization occurs. MBean Attribute: JTAMBean.BeforeCompletionIterationLimit Minimum value: 1
Max Transactions	The maximum number of simultaneous in-progress transactions allowed on a server in this WebLogic Server domain. MBean Attribute: JTAMBean.MaxTransactions Minimum value: 1 Maximum value: 2147483647

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Configuration Options for JTA (transactions)

Name	Description
Max Unique Name Statistics	The maximum number of unique transaction names for which statistics are maintained. Separate statistics are maintained for up to the configured number of unique transaction names. Additional names are aggregated under `weblogic.transaction.statistics.namedOverflow`. The default limit is 1000. A transaction name typically represents a category of business transactions, such as "funds-transfer." MBean Attribute: JTAMBean.MaxUniqueNameStatistics Minimum value: 0 Maximum value: 2147483647
Checkpoint Interval Seconds	The interval at which the transaction manager creates a new transaction log file and checks all old transaction log files to see if they are ready to be deleted. MBean Attribute: JTAMBean.CheckpointIntervalSeconds Minimum value: 10 Maximum value: 1800
Write recovery logs when determiners configured	Indicates two-phase transaction recovery logs are written even if one or more determiners are configured. MBean Attribute: JTAMBean.TLOGWriteWhenDeterminerExistsEnabled
Determiners	Select a transaction resource (determiner) from the list of resources. For JMS, select WebLogic JMS as the determiner. When a determiner is configured, the determiner's in-doubt transaction records are used during transaction recovery. The determiner-based NoTLog feature is deprecated as of Oracle WebLogic Server 26.1.0.0.0. See XA Transactions without Transaction TLog Write. MBean Attribute: JTAMBean.Determiners
Forget Heuristics	Specifies whether the transaction manager automatically performs an XA Resource forget operation for heuristic transaction completions. When enabled, the transaction manager automatically performs an XA `Resource forget()` operation for all resources as soon as the transaction learns of a heuristic outcome. Disable this feature only if you know what to do with the resource when it reports a heuristic decision. MBean Attribute: JTAMBean.ForgetHeuristics

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Configuration Options for JTA (transactions)

Name	Description
Unregister Resource Grace Period	The amount of time, in seconds, a transaction manager waits for transactions involving the resource to complete before unregistering a resource. This grace period helps minimize the risk of abandoned transactions because of an unregistered resource, such as a JDBC data source module packaged with an application. During the specified grace period, the `unregisterResource` call blocks until the call returns and no new transactions are started for the associated resource. If the number of outstanding transactions for the resource goes to 0, the `unregisterResource` call returns immediately. At the end of the grace period, if outstanding transactions are associated with the resource, the `unregisterResource` call returns and a log message is written to the server on which the resource was previously registered. MBean Attribute: JTAMBean.UnregisterResourceGracePeriod Minimum value: 0 Maximum value: 2147483647
Execute XA Calls In Parallel	Indicates that XA calls are executed in parallel if there are available threads. MBean Attribute: JTAMBean.ParallelXAEnabled
Enable Two Phase Commit	Indicates that the two-phase commit protocol is used to coordinate transactions across two or more resource managers. If not selected: • Two-phase commit is disabled and any attempt to use two-phase commit results in a `RollbackException` being thrown. • All transaction logging is disabled, including checkpoint records. MBean Attribute: JTAMBean.TwoPhaseEnabled Changes take effect after you redeploy the module or restart the server
Enable Tightly Coupled Transactions	Indicates tight coupling of transaction branches that span different transaction manager systems. When enabled, WebLogic uses the transaction identifier of a transaction imported by the `InterposedTransactionManager` for XA calls rather than an internally mapped Xid. This applies to inter-domain WebLogic transactions and transactions imported from Tuxedo. This allows for tight coupling of transaction branches for transactions that span across different transaction manager systems. If a transaction between WebLogic and Tuxedo resources uses a GridLink Data Source with GridLink Affinity enabled, the XA Affinity context is automatically used for the transaction. MBean Attribute: JTAMBean.TightlyCoupledTransactionsEnabled

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Configuration Options for JTA (transactions)

Name	Description
Enable Cluster-Wide Recovery	Indicates that cluster-wide recovery is used for distributed transactions. When enabled, recovery operations for a distributed transaction are applied to all the servers of the cluster hosting a `InterposedTransactionManager` rather than just the server hosting the `InterposedTransactionManager`. MBean Attribute: JTAMBean.ClusterwideRecoveryEnabled

- Unregister Resource Grace Period
- Additional Attributes for Managing Transactions
- XA Transaction Cluster Affinity

Unregister Resource Grace Period

If you have resources that you may occasionally undeploy and redeploy such as a JDBC data source module packaged with an application, minimize the risk of abandoned transactions because of an unregistered resource by setting the Unregistered Resource Grace Period for the domain. The grace period is the number of seconds that the transaction manager waits for transactions to complete before unregistering a resource.

During the specified grace period, the `unregisterResource` call blocks until the call returns, and no new transactions are started for the associated resource. If the number of outstanding transactions for the resource goes to 0, the unregisterResource call returns immediately.

At the end of the grace period, if there are still outstanding transactions associated with the resource, the unregisterResource call returns and a log message is written on the server on which the resource was previously registered.

Additional Attributes for Managing Transactions

By default, if an XA resource that is participating in a global transaction fails to respond to an XA call from the WebLogic Server transaction manager, WebLogic Server flags the resource as unhealthy and unavailable, and blocks any further calls to the resource in an effort to preserve resource threads. The failure can be caused by either an unhealthy transaction or an unhealthy resource—there is no distinction between the two causes. In both cases, the resource is marked as unhealthy. To mitigate this limitation, WebLogic Server provides the following configuration attributes:

- `Maximum Duration of XA Calls`—Sets the maximum allowed duration (in milliseconds) of XA calls to XA resources.
- `Maximum Duration XA Resource Unavailable`—The maximum duration (in milliseconds) that an XA resource is marked as unhealthy.
- `Maximum Resource Requests on a Server`—Maximum number of concurrent requests to resources allowed for each server in the domain.

These attributes are configurable at the domain and the cluster level.

Copyright © 2007, 2026, Oracle and/or its affiliates.

XA Transaction Cluster Affinity

XA transaction affinity allows server instances that are participating in a global transactions to service related requests rather than load-balancing these requests to other member servers. When `Enable Transaction Affinity=true`, cluster throughput can be increased by:

- Reducing inter-server transaction coordination traffic
- Improving resource utilization, such as reducing JDBC connections
- Simplifying asynchronous processing of transactions

If the cluster does not have a member participating in the transaction, the request is load balanced to an available cluster member. If the selected cluster member fails, the JTA Transaction Recovery Service can be migrated using the Roadmap for Configuring Automatic Migration of the JTA Transaction Recovery Service in *Administering Clusters for Oracle WebLogic Server*.

See Configure Clusters in *Oracle WebLogic Remote Console Online Help*. You can also enable XA transaction affinity on the command line using `-Dweblogic.cluster.TxnAffinityEnabled=true`.

Configuring Network Channels for JTA Communication

Network channels are a configurable resource in WebLogic Server that you can configure to separate and load balance network traffic. Also, you can configure network channels to separate internal server communication from client user communication.

WebLogic Server automatically generates the default network channel. In place of the default network channel, you can configure custom network channels for JTA communications. The WebLogic Server transaction manager uses these custom network channels for coordinating distributed transactions that update multiple servers.

The network channels specified for JTA communication support mixed internal and external addressing for cross-domain JTA communication and interoperability with servers configured with the default network channel. A global transaction context propagates between servers when an application accesses remote objects such as, EJBs and resources such as, JMS and JDBC. Each server adds its local address information into the transaction context so that it can be used by other servers to communicate with it. The transaction context contains the address information of the coordinating server and all the subordinate servers.

During server initialization, the coordinator URL, which represents the local server, is constructed using the address information from the static configuration and runtime overrides. The coordinator URL of the local server is added to the transaction context at coordinator assignment time or when the transaction propagation context is received on a remote server. For a remote coordinator assignment, the coordinator URL is constructed from information about the remote server obtained from the RMI connection and added to the propagation context during the outbound RMI call.

You can create up to four types of URLs for JTA interserver communication that are obtained from the server network channel configuration:

- Primary
- Secure
- Public
- Public Secure

Copyright © 2007, 2026, Oracle and/or its affiliates.

The created URLs are stored with the JTA descriptor that is used to identify coordinator and subordinate servers, are propagated between servers, and are persisted with the transaction context and in server checkpoint records. When a server participant needs to obtain the coordinator or sub-coordinator remote object for a remote participant, one of the four URLs are used to establish the JNDI initial context and perform the lookup.

See Understanding Network Channels and Configuring a Channel in *Administering Server Environments for Oracle WebLogic Server* for more information about configuring a network channel. Use the WebLogic Remote Console or `NetworkAccessPointMBean` to configure a network channel and set the `protocol`, `listen address`, `listen port`, `public address`, and the `public port` attributes to construct the JTA coordinator URL.

Constructing the Server URLs

The following example shows the configuration of a server that listens for `t3` traffic on the network address corresponding to ports `7001` and `7101`:

Example 2-1 Server Configuration

```
<server>
  <name>myserver</name>
  <listen-port>7001</listen-port>
  <listen-port-enabled>true</listen-port-enabled>
  <listen-address>myhost</listen-address>
  <network-access-point>
    <name>channelt3</name>
    <protocol>t3</protocol>
    <listen-address>myhost</listen-address>
    <public-address>publicaddress</public-address>
    <listen-port>7101</listen-port>
    <public-port>7101</public-port>
  </network-access-point>
</server>
```

The coordinator URL is derived from the network channel configuration and is of the form `<protocol>://host:port`. For the network configuration of the server defined in the example, the supported URLs are `t3://myhost:7001`, `t3://myhost:7101`, and `t3://publicaddress:7101`.

The server level attributes that are supported for configuring JTA network channel are:

- `ServerMBean.TransactionPrimaryChannelName` – The name of the server network channel that is used to derive the primary URL for internal JTA communication with the server. If not set, then the primary URL is initialized using the server's default channel configuration.

  Table 2-2 Example of Network Channel with Primary URL

Protocol	listen-address:listen-port	public-address:public-port	Primary URL
t3	host:7001	NA	t3://host:7001
t3	host:7001	public-address:7201	t3://public-address:7201

- `ServerMBean.TransactionSecureChannelName` – The name of the server network channel that is used in secure communication. The network access point configuration must have the `t3s` protocol. If this attribute is not specified, then the default secure channel is used for

Copyright © 2007, 2026, Oracle and/or its affiliates.

secure URL if enabled. Secure channel takes precedence over administration channel configuration.

Table 2-3 Example of Network Channel with Secure URL

Protocol	listen-address:listen-port	public-address:public-port	Secure URL
t3s	host:9003	NA	t3s://host:9003
t3s	host:9003	public-address:9003	t3s://public-address:9003

- ServerMBean.TransactionPublicChannelName – The name of the server network channel that is used to determine the public URL. It is an optional attribute.

Table 2-4 Example of Network Channel with Public URL

Protocol	listen-address:listen-port	public-address:public-port	Public URL
t3	host:7001	NA	t3://host:7001
t3	host:7001	public-address:7201	t3://public-address:7201

- ServerMBean.TransactionPublicSecureChannelName – The name of the server network channel that is used to derive the public secure URL. The referenced network access point configuration must have the t3s protocol. It is an optional attribute.

Table 2-5 Example of Network Channel with Public Secure URL

Protocol	listen-address:listen-port	public-address:public-port	Public Secure URL
t3s	host:9303	NA	t3s://host:9303
t3s	host:9303	public-address:9333	t3s://public-address:9333

> **Note**
>
> The network channel configuration can contain only one public secure and public nonsecure addresses. Multiple public addresses are not supported.

Conditions for Constructing the Server URL

Depending on whether the listen-address or public-address is set, the URL for primary, public, secure, and public secure is constructed. If you configure the same network access point configuration for primary and public or secure and public secure, then the listen-address:listen-port is used for primary and secure URL while the public-address:public-port is used for public and public secure URL.

The following table shows an example of how the URL is constructed from the network access point attributes:

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-6 Conditions for Constructing the Primary and Public URLs

Protocol	listen-address:listen-port	public-address:public-port	Primary URL	Public URL
t3	host:7001	public-address:7201	t3://host:7001	t3://public-address:7201

Table 2-7 Conditions for Constructing the Secure and Public Secure URLs

Protocol	listen-address:listen-port	public-address:public-port	Secure URL	Public Secure URL
t3s	host:9303	public-address:9333	t3s://host:9303	t3s://public-address:9333

Using Public Addresses for Remote Domains

To communicate with a remote domain using the public URL, use the new attribute `JTAMBean.UsePublicAddressesForRemoteDomains` defined with the domain names.

If the target domain name is not specified in the attribute, then the primary or secure URL is used.

Using Nonsecure Addresses for Domains

To specify the domains for which only nonsecure URLs must be used for JTA communication, use the new attribute `JTAMBean.UseNonSecureAddressesForDomains` defined with the domain names.

> **Note**
>
> Public addresses are used only for cross-domain communication.

Interoperability with Servers Using Default Channels

Mixed configurations are supported to use network channels such as:

- Configuring the servers in a domain to use the network channel for JTA communication.
- Using the server default channel based on listen address and port, or listen address and SSL port.
- Using the primary URL when servers need to connect to a remote server that is configured to use the default channel.

In the example server configuration, Example 2-1, if the `ServerMBean.TransactionPublicChannelName` is set to `channelt3`, then the resulting primary URL will be `t3://myhost:7001` and the public URL will be `t3://publicaddress:7101`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using Transaction Log Files to Recover Transactions

Each server has a transaction log which stores information about committed transactions coordinated by the server that may not have been completed. WebLogic Server uses the transaction log (TLogs) when recovering from system crashes or network failures.

You cannot directly view the transaction log—the records are in a binary format and are stored in either the default persistent store or a JDBC TLog store for the server.

- Using the Default Persistent Store
- Using a JDBC TLOG Store
- Last Logging Resource
- XA Transactions without Transaction TLog Write

Using the Default Persistent Store

To take advantage of the migration capability of the Transaction Recovery Service for servers in a cluster, you must store the transaction log in a location that is available to a server and its backup servers, preferably on a dual-ported SCSI disk or on a Storage Area Network (SAN). See Setting the Path for the Default Persistent Store.

If the file system on which the default store saves transaction log records runs out of space or is inaccessible, `commit()` throws `SystemException`, and the transaction manager places a message in the system error log. No transactions are committed until more space is available.

- Setting the Path for the Default Persistent Store
- Setting the Default Persistent Store Synchronous Write Policy

Setting the Path for the Default Persistent Store

Each server instance, including the administration server, has a default persistent store, which is a file-based store that is available to subsystems that do not require explicit selection of a particular store and function best by using the system's default storage mechanism. The transaction manager uses the default persistent store to store transaction log records. In many cases, the default persistent store requires no configuration. However, to enable migration of the Transaction Recovery Service, you must configure the default persistent store so that it stores its data files on a persistent storage solution that is available to other servers in the cluster if the original server fails.

Setting the Default Persistent Store Synchronous Write Policy

WebLogic Server uses the default persistent store to store transaction log records. Select a write policy for the default store to change the way WebLogic Server writes records to disk, see Guidelines for Configuring a Synchronous Write Policy.

Using a JDBC TLOG Store

You can configure a JDBC TLog store to persist transaction logs to a database, which allows you to leverage replication and HA characteristics of the underlying database, simplify disaster recovery, and improve Transaction Recovery service migration. See Administering the Persistent Store.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Last Logging Resource

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. See Logging Last Resource Transaction Optimization.

XA Transactions without Transaction TLog Write

As described in Using Transaction Log Files to Recover Transactions, TLogs are used to resolve in-doubt transactions. However, there can be a significant cost associated with creating and maintaining these files, such as read/write costs, network calls between resource managers, and HA storage.

> **Note**
>
> Server and resource checkpoints in a XA transaction are still written to the TLog. These checkpoints are written when the resources are first involved in a global transaction, updated only if there are changes to the transaction participants, and purged only if they are no longer used or become unavailable.

The following section provides information on how to improve XA transaction performance by eliminating TLogs when XA transactions span a single transaction manager (TM):

- What is a Determiner Resource?
- Configuring XA Transactions without TLogs

What is a Determiner Resource?

Determiners are JDBC XA and JMS XA resources whose in-doubt transaction records are used during transaction recovery when a TLog is not present.

- Best Practices When Using Determiner Resources

Best Practices When Using Determiner Resources

The following section provides information on best practices for using determiner resources in global transactions:

- The determiner resource must be targeted to the server where the global transaction executes. In a clustered environment, target the determiner resource to the cluster to ensure availability during whole server migration.
- Configure the minimum number of determiner resources possible. This reduces the number of resources that must be available for recovery.

Configuring XA Transactions without TLogs

The following sections provide information on how to configure, update, and remove determiners in your environment.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Before you start, review Limitations and Considerations When Configuring Transactions without TLogs

- How to Configure a Determiner
- How to Remove a Determiner
- Limitations and Considerations When Configuring Transactions without TLogs

How to Configure a Determiner

To configure a determiner:

1. Ensure that all XA transactions are complete on the server or servers that will host the JDBC XA or JMS XA resources you want to configure as a determiner resource.
2. Stop any server or cluster hosting JDBC XA resources to be used as a determiner.
3. You can configure a determiner for a domain or for a cluster.
   - For a domain: In the edit tree select **Services** > **JTA**.
   - For a cluster: In the edit tree, select the cluster you want to configure and then select the **JTA** tab.
4. Specify a list of one or more determiners separated by line breaks in the **Determiner** attribute.

   JDBCA XA resource determiner names have the pattern <*XAdsname*>+"_"+<*dname*> , where *XAdsname* is the name of the data source and *dname* is the name of the domain. For example:

   ```
   myDS1_myDomain
   myDS2_myDomain
   ```

   JMS XA resource determiner names have the pattern "WLStore_"+<*dname*>+"_"+<*JMSXAResource*>, where *JMSXAResource* is the name of the JMS persistent store and *dname* is the name of the domain. If no persistent store is configured, default store is used and the naming convention is "WLStore_"+<*dname*>"+"_"+"_"+"WLS_"+<*server name*>, where *server name* is the name of the server and *dname* is the name of the domain. For example:

   ```
   WLStore_myDomain_myStore1
   WLStore_myDomain_myStore2
   WLStore_XADomain__WLS_myServer
   ```

   Note: Target the determiner resource to ensure high availability, see Best Practices When Using Determiner Resources.
5. Optionally, enable the `Write recovery logs when determiners configured` attribute to write transaction recovery logs even if one or more determiners are configured.
6. Click **Save**.
7. To activate these changes, click **shopping cart** on top right corner and select **commit changes**.

   Not all changes take effect immediately—some require a restart.
8. Restart any server or cluster hosting a configured determiner resource.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Your environment is now ready to resume transaction processing.

How to Remove a Determiner

To remove a determiner:

1. Ensure that all XA transactions are complete on the server or servers that host the JDBC XA or JMS XA resources you want to remove as a determiner resource.
2. Stop any server or cluster hosting determiner resources to be removed.
3. You can remove a determiner for a domain or for a cluster.
   - For a domain: In the edit tree select **Services** > **JTA**.
   - For a cluster: In the **Edit Tree** select the cluster you want to configure and select the **JTA** tab.
4. Remove one or more determiners from the list specified in the **Determiner** attribute.
5. Click **Save**.
6. To activate these changes, click **shopping cart** on top right corner and select **commit changes**.

   Not all changes take effect immediately—some require a restart.
7. Restart any server or cluster shut down in Step 2.

Your environment is now ready to resume transaction processing.

Limitations and Considerations When Configuring Transactions without TLogs

The following section provides important information on limitations and considerations when configuring transactions without TLogs:

- Configuring Transactions without TLogs is not supported when:
  - More than one TM participates in global transactions
  - Global transactions include JTS or LLR resources
  - Participating transactions include WS-AT, OTS, WLS/Tuxedo transactions, and transactions that include foreign transaction managers
  - Only read operations are executed in the determiner resource as part of the global transaction work.
  - `parallel-xa-enabled` is set to `false` (default value is `true`).
- Server and resource checkpoints are still written to the TLOG. Checkpoints are written when the resources are first involved in a global transaction, updated only if there are changes to the transaction participants, and purged only if they are no longer used or become unavailable. Because checkpoints are created early in the transaction, as long as there are no changes to the participants in the global transactions, there is little danger of a checkpoint being out of sync during transaction recovery spanning multiple sites or data centers.
- `Abandon Timeout Seconds` is not supported. Recovery is attempted on in-doubt transactions until they are resolved or the resource is administratively removed.
- WebLogic Server will not start if a determiner resource is not available because XA transactions cannot be recovered.
- To change the targeting of a determiner resource, you need to:

Copyright © 2007, 2026, Oracle and/or its affiliates.

1. Ensure that all XA transactions are complete on the server or servers that host the determiner resource.
2. Stop any server or servers hosting the determiner resource.
3. Retarget the determiner resource.
4. Start any server or cluster hosting the determiner resource.

- If more than one server is participating in a global transaction, a transaction entry is written to the TLog.
- If more than one determiner resource is available from the list of specified determiners, the first determiner enlisted is implicitly nominated as the determiner of the transaction.
- Applications can have a combination of global transactions, some that enlist a determiner resource and some that do not. Only those transactions that do not enlist the determiner resource will be written to the TLog.
- This feature does not support Dynamic Clusters.
- This feature supports Whole Server Migration (WSM), see Whole Server Migration in *Administering Clusters for Oracle WebLogic Server*. Service migration is not supported with determiner resources.

Read-Only, One-Phase Commit Optimizations

When resource managers, such as the Oracle Database (including AQ and RAC), provide read-only optimizations, Oracle WebLogic can provide a read-only, one-phase commit optimization. Read-only, one-phase commit optimization provides a number of benefits – even when enabling multiple connections of the same XA transactions – such as eliminating `XAResource.prepare` network calls and transaction log writes, both in Oracle WebLogic and in the resource manager.

> **Note**
>
> Read-only, One-phase Commit Optimization requires Oracle Database 11.1.0.7.3PSU or later.

For applications that do not require two-phase commit transactions, you can further optimize performance by also disabling the WebLogic "Two Phase Commit" protocol, which coordinates transactions across two or more resource managers. Disabling two-phase commits, does the following:

- Removes persistent in-doubt logging and locks, as well as bookkeeping overhead in the database.
- Removes all checkpoint logging in WebLogic.
- Enforces and/or tests the assumption that a particular server instance does not require two-phase commit.
- Removes the need for WebLogic migration (whole server or service) recovery, which in turn removes the need for additional assets/capacity, management, etc., involved in such migrations.
- Configuring Read-only, One-phase Commit Optimization and Two-phase Commit Disablement
- Monitoring Read-only, One-phase Transaction Statistics

Copyright © 2007, 2026, Oracle and/or its affiliates.

Configuring Read-only, One-phase Commit Optimization and Two-phase Commit Disablement

In order to enable the read-only, one-phase commit optimization and disable two-phase commits, configure the following JTA domain configuration attributes:

- Execute XA Calls In Parallel – Set to `false` to enable the read-only, one-phase commit optimization.
- Enable Two Phase Commit – Optionally, set to `false` to disable two-phase commit transactions. This disables all transaction logging, including checkpoint records. Any attempt to use two-phase commit will result in a RollbackException being thrown.

  Important! The Enable Two Phase Commit setting, which is `true` by default, should not to be set to `false` unless it is well-known that the application only uses a resource manager that provides read-only optimization, such as Oracle database, or that the application only uses a single connection to a single resource manager.

> **Note**
>
> If a XA resource returns an `XA_OK` vote from a prepare (for example, if it is not an Oracle database), and the WebLogic instance then crashes before rollback can take place, there will be an in-doubt record and locks will be held in the resource manager (database) that will need to be manually resolved.

For more information on all JTA domain configuration options, see Configure Domain JTA Options in the *Oracle WebLogic Remote Console Online Help*.

Monitoring Read-only, One-phase Transaction Statistics

For monitoring purposes, there are five transaction processing statistics on the JTA Monitoring page, which together break down the Transaction Committed Total Count statistic to better track any read-only, one-phase commit transactions.

- Transaction No Resources Committed Total Count – The total number of transactions with no enlisted resources that were committed since the server was started.
- Transaction One Resource One Phase Committed Total Count – The total number of transactions with only one enlisted resource that were one-phase committed since the server was started.
- Transaction Read Only One Phase Committed Total Count – The total number of transactions with more than one enlisted resource that were one-phase committed due to read-only optimization since the server was started.
- Transaction Two Phase Committed Total Count – The total number of transactions with more than one enlisted resource that were two-phase committed since the server was started.
- Transaction LLR Committed Total Count – The total number of LLR transactions that were committed since the server was started.

  Note: If the only resource enlisted in a JTA transaction is an LLR data source, then such transactions are included under the Transaction One Resource One Phase Committed Total Count category rather than the Transaction LLR Committed Total Count category.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3

Managing Transactions

Learn about administration tasks that are used to manage transactions. These tasks include monitoring transactions, handling heuristic completions, how to abandon a transaction, resolving in-flight transactions, and transaction recovery.
Monitor transactions on a server using statistics and monitoring facilities. Use the Remote Console to configure these features and to display the resulting output.

- Monitoring Transactions
  Monitor transactions for each server in the domain using the Remote Console. Transaction statistics are displayed for a specific server, not the entire domain.
- Handling Heuristic Completions
- Moving a Server
  A server instance is identified by its URL (IP address or DNS name plus the listening port number). Changing the URL by moving the server to a new machine or changing the Listening Port of a server on the same machine effectively moves the server so the server identity may no longer match the information stored in the transaction logs.
- Abandoning Transactions
  You can choose to abandon incomplete transactions after a specified amount of time.
- Manually Resolving Current (Inflight) Transactions
  If a transaction does not complete normally due to system or network failures, there may be locks held on behalf of the pending transaction that may inhibit the progress of other transactions. You can wait for the WebLogic Server Transaction Manager to remove the transaction from its internal data structures, or you can manually resolve "stuck" transactions.
- Transaction Recovery After a Server Fails
  The WebLogic Server transaction manager is designed to recover from system crashes with minimal user intervention. The Transaction Recovery Service provided by WebLogic Server makes every effort to resolve transaction branches that are prepared by resource managers with a commit or roll back, even after multiple crashes or crashes during recovery.

Monitoring Transactions

Monitor transactions for each server in the domain using the Remote Console. Transaction statistics are displayed for a specific server, not the entire domain.

For instructions, see the following pages in the *Oracle WebLogic Remote Console Online Help*:

- View Transaction Statistics
- View Current Transactions

Handling Heuristic Completions

A **heuristic completion** (or heuristic decision) occurs when a resource makes a unilateral decision during the completion stage of a distributed transaction to commit or rollback updates.

Copyright © 2007, 2026, Oracle and/or its affiliates.

This can leave distributed data in an indeterminate state. Network failures or resource timeouts are possible causes for heuristic completion.
In the event of an heuristic completion, one of the following heuristic outcome exceptions may be thrown:

- `HeuristicRollback`—one resource participating in a transaction decided to autonomously rollback its work, even though it agreed to prepare itself and wait for a commit decision. If the Transaction Manager decided to commit the transaction, the resource's heuristic rollback decision was incorrect, and might lead to an inconsistent outcome since other branches of the transaction were committed.
- `HeuristicCommit`—one resource participating in a transaction decided to autonomously commit its work, even though it agreed to prepare itself and wait for a commit decision. If the Transaction Manager decided to rollback the transaction, the resource's heuristic commit decision was incorrect, and might lead to an inconsistent outcome since other branches of the transaction were rolled back.
- `HeuristicMixed`—the Transaction Manager is aware that a transaction resulted in a mixed outcome, where some participating resources committed and some rolled back. The underlying cause was most likely heuristic rollback or heuristic commit decisions made by one or more of the participating resources.
- `HeuristicHazard`—the Transaction Manager is aware that a transaction might have resulted in a mixed outcome, where some participating resources committed and some rolled back. But system or resource failures make it impossible to know for sure whether a Heuristic Mixed outcome definitely occurred. The underlying cause was most likely heuristic rollback or heuristic commit decisions made by one or more of the participating resources.

When an heuristic completion occurs, a message is written to the server log. Refer to your database vendor documentation for instructions on resolving heuristic completions.

Some resource managers save context information for heuristic completions. This information can be helpful in resolving resource manager data inconsistencies. If the `ForgetHeuristics` attribute is selected (set to true) on the JTA panel of the WebLogic Remote Console, this information is removed after an heuristic completion. When using a resource manager that saves context information, you may want to set the `ForgetHeuristics` attribute to false.

Moving a Server

A server instance is identified by its URL (IP address or DNS name plus the listening port number). Changing the URL by moving the server to a new machine or changing the Listening Port of a server on the same machine effectively moves the server so the server identity may no longer match the information stored in the transaction logs.

- If the new server has the same URL as the old server, the Transaction Recovery Service searches all transaction log files for incomplete transactions and completes them as described in Transaction Recovery Service Actions After a Crash.
- When the coordinator server is in the same domain as the sub-coordinator and the server URL changes, the coordinator queries the Administration Server for the new URL of the sub-coordinator and the propagation of any new transactions and any transactions that are committing or rolling back use the new URL. Transaction branches for the sub-coordinator with pending commit records stored in the coordinator's transaction log files before the URL change are unrecoverable. If you wish, you can delete the transaction log files of the coordinator. This step prevents the Transaction Recovery Service from attempting to resolve these transactions until the value of the `AbandonTimeoutSeconds` parameter is exceeded. See Abandoning Transactions and How to Remove Transaction Records.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- When transactions span multiple domains and if a server acting as a remote transaction sub-coordination fails and its URL changes, any ongoing transactions do not complete (commit or are rolled back) because the coordinator is unable to communicate with the remote domain's Admin server. The coordinator is unable to contact the sub-coordinator using the new URL and any ongoing transactions fail. The coordinator attempts the commit or rollback request until the `AbandonTimeoutSeconds` value is exceeded. See Abandoning Transactions . Any new transactions fail because the coordinator cannot contact the sub-coordinator. The TLogs of the coordinator and sub-coordinators, excluding the moved server domain, must be deleted. See How to Remove Transaction Records.

Oracle recommends configuring server instances using DNS names rather than IP addresses to promote portability.

If you move a server to a new machine, follow the instructions for Recovering Transactions For a Failed Non-Clustered Server.

Abandoning Transactions

You can choose to abandon incomplete transactions after a specified amount of time.

In the two-phase commit process for distributed transactions, the transaction manager coordinates all resource managers involved in a transaction. After all resource managers vote to commit or rollback, the transaction manager notifies the resource managers to act—to either commit or rollback changes. During this second phase of the two-phase commit process, the transaction manager continues to try to complete the transaction until all resource managers indicate that the transaction is completed. Using the `AbandonTimeoutSeconds` attribute, set the maximum time, in seconds, that a transaction manager persists in attempting to complete a transaction during the second phase of the commit protocol. The default value is 86400 seconds, or 24 hours. After the abandon transaction timer expires, no further attempt is made to resolve the transaction with any resources that are unavailable or unable to acknowledge the transaction outcome. If the transaction is in a prepared state before being abandoned, the transaction manager rolls back the transaction to release any locks held on behalf of the abandoned transaction and writes an heuristic error to the server log.

For instructions on how to set the `AbandonTimeoutSeconds` attribute, see Configure Domain JTA Options in the *Oracle WebLogic Remote Console Online Help*.

> **Note**
>
> It is not recommended to change the default `AbandonTimeoutSeconds` attribute to a small value. When a global transaction is abandoned, resource manager transaction branches could be left pending, which would require manual intervention to resolve.

- Transaction Completion Timeout

Transaction Completion Timeout

When an application or server component instructs a transaction to commit or rollback, the commit or rollback API will block while the transaction manager processes the two-phase commit protocol. If the transaction is unable to complete within a period of time (up to two minutes by default) a `SystemException` will be raised. A `SystemException` is non-deterministic relative to transaction outcome so an application environment must provide special exception handling for this case which often involves manually analyzing the transaction activity and state

Copyright © 2007, 2026, Oracle and/or its affiliates.

of the resources involved in the transaction. The period of time waiting for a transaction to complete can be changed by setting the `completion-timeout-seconds` attribute.

For configuration information, see:

- Configure Domain JTA Options in *Oracle WebLogic Remote Console Online Help*
- CompletionTimeoutSeconds in *MBean Reference for Oracle WebLogic Server*

> **Note**
>
> The *completion-timeout-seconds* attribute does not apply to imported transactions such as JCA transactions or to recovering transactions.

If transactions are taking a long time to complete the two-phase commit protocol then it could indicate a problem with server or resource manager participants. Viewing the current active transactions and server statistics on each of the server participants could help identify the cause of the transaction completion delays, see Monitoring Transactions.

Manually Resolving Current (Inflight) Transactions

If a transaction does not complete normally due to system or network failures, there may be locks held on behalf of the pending transaction that may inhibit the progress of other transactions. You can wait for the WebLogic Server Transaction Manager to remove the transaction from its internal data structures, or you can manually resolve "stuck" transactions.

After the Abandon Timeout period has elapsed, the WebLogic Server Transaction Manager removes the transaction from its internal data structures and writes a heuristic error to the server log. However, to manually resolve a transaction, you view current (inflight) transactions for a server (see View Current Transactions in *Oracle WebLogic Remote Console Online Help*) and then view details about a specific transaction by clicking the transaction id. You can then force a commit or a rollback, depending on the status of the transaction.

> **Note**
>
> It is possible for a transaction to have different states at different servers. For instance, a transaction may have been committed at the coordinating server, but a remote participant may not have received the commit instruction.

The following table provides information on transaction status and resolution options.

Table 3-1 Transaction Status Definitions and Manual Resolution Options

Status	Definition	Forced Commit?	Forced Rollback?
Active	The application is processing the transaction. The transaction has not yet reached the two-phase commit processing.		Y

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 3-1 (Cont.) Transaction Status Definitions and Manual Resolution Options

Status	Definition	Forced Commit?	Forced Rollback?
Preparing	Corresponds to the interval between when the transaction manager starts the jakarta.transaction.Synchronization beforeCompletion() callback processing, through the first phase of the 2PC protocol, and up to the point when all participants have responded, "ready to commit."		Y
Prepared	The interval between when all participants have responded to prepare up to the commit point (commit log record is flushed to disk) or to the initiation of rollback processing.	Y	Y
Committing	The time from when the commit decision is made up to the point when all participants have been informed of the outcome and the `jakarta.transaction.Synchronization afterCompletion()` callback processing has completed.	Y	
Committed	The transaction has been committed. It is likely that heuristics exists, otherwise the transaction would have been completed and would not have been displayed in the list of current transactions.	Y	
Rolling Back	This state occurs from the point when rollback processing is initiated up to the point when all participants have been instructed to rollback and the `jakarta.transaction.Synchronization afterCompletion()` callback processing has completed.		Y
Rolled Back	The transaction has been rolled back. It is likely that heuristics exists, otherwise the transaction would have been destroyed and would not have been displayed in the list of current transactions.		Y
Marked Roll Back	The transaction has been marked for rollback, perhaps as a result of a setRollbackOnly operation.		Y
No Transaction			
Unknown	Current status cannot be determined.	Y	Y

- Manual Commit and Rollback Options

Manual Commit and Rollback Options

To manually resolve a transaction, you can choose from the following options. Options are restricted as described in Table 3-1.

- **Force Local Commit**—Each participating resource that is registered on the server is issued a commit operation for the specified transaction and the transaction will be removed from the local transaction manager's data structures. If the local server is the coordinator for the transaction, the commit record is released.
- **Force Global Commit**—A local commit operation is attempted at each participating server for the specified transaction. If this option is invoked on a non-coordinating server, the coordinator will be contacted to process the operation. The coordinating server will issue asynchronous requests to each participant server.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- **Force Local Rollback**—Each participating resource that is registered on the local server is issued a rollback operation for the specified transaction. The transaction will then be removed from the local transaction manager's data structures.
- **Force Global Rollback**—A local rollback operation is attempted at each participating server for the specified transaction. If this option is invoked on a non-coordinating server, the coordinator will be contacted to process the operation. The coordinating server will issue asynchronous requests to each participant server.

> **Note**
>
> When you select any of these options, WebLogic Server writes entries to the server log.

The difference between the Local and Global options is that Local options act only upon the current server resources (resources on the server that you select in the navigation tree in the left pane of the WebLogic Remote Console), whereas the Global options attempt to perform the operation across all participating servers. If a Global operation is invoked for a transaction that is not coordinated by the local server then an attempt will be made to contact the coordinator of the transaction in order to perform the operation. If the coordinator cannot be reached, the operation will fail with a `jakarta.transaction.SystemException`.

In the case where a transaction may have been committed at the coordinating server (committing status), but a remote participant did not receive the commit instruction (prepared status). You can force a local commit on the remote participant to complete the transaction. In this case it is possible to force a rollback on the remote participant since its transaction state will still be prepared, but the transaction will complete heuristically. If you try to force a global rollback, the operation will fail because the state at the coordinator is committing. You cannot roll back a transaction with the committing status.

Transaction Recovery After a Server Fails

The WebLogic Server transaction manager is designed to recover from system crashes with minimal user intervention. The Transaction Recovery Service provided by WebLogic Server makes every effort to resolve transaction branches that are prepared by resource managers with a commit or roll back, even after multiple crashes or crashes during recovery.

To facilitate recovery after a crash, the Transaction Recovery Service automatically attempts to recover transactions on system startup. On startup, the Transaction Recovery Service parses all transaction log records for incomplete transactions and completes them as described in Transaction Recovery Service Actions After a Crash.

Because the Transaction Recovery Service is designed to gracefully handle transaction recovery after a crash, Oracle recommends that you attempt to restart a crashed server and allow the Transaction Recovery Service to handle incomplete transactions.

If a server crashes and you do not expect to be able to restart it within a reasonable period of time, you may need to take action. Procedures for recovering transactions after a server failure differ based on your WebLogic Server environment. For a non-clustered server, you can manually move the server (with the default persistent store DAT file) to another system (machine) to recover transactions. See Recovering Transactions For a Failed Non-Clustered Server . For a server in a cluster, you can manually migrate the whole server or the *Transaction Recovery Service* to another server in the same cluster. Migrating the Transaction Recovery Service involves selecting a server with access to the transaction logs to recover transactions, and then migrating the service using the WebLogic Remote Console or the WebLogic command-line interface.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> For non-clustered servers, you can only move the entire server to a new system. For clustered servers, you can migrate the entire server or temporarily migrate the Transaction Recovery Service.

For more information about migrating the Transaction Recovery Service, see Recovering Transactions For a Failed Clustered Server. For more information about clusters, see *Administering Clusters for Oracle WebLogic Server*.

The following sections provide information on how to recover after a failure:

- Transaction Recovery Service Actions After a Crash
- Recovering Transactions For a Failed Non-Clustered Server
- Recovering Transactions For a Failed Clustered Server
- How to Remove Transaction Records

Transaction Recovery Service Actions After a Crash

When you restart a server after a crash or when you migrate the Transaction Recovery Service to another (backup) server, the Transaction Recovery Service does the following:

- Complete transactions ready for second phase of two-phase commit

  For transactions for which a commit decision has been made but the second phase of the two-phase commit process has not completed (transactions recorded in the transaction log), the Transaction Recovery Service completes the commit process.

- Resolve prepared transactions

  For transactions that the transaction manager has prepared with a resource manager (transactions in phase one of the two-phase commit process), the Transaction Recovery Service must call `XAResource.recover()` during crash recovery for each resource manager and eventually resolve (by calling the `commit()`, `rollback()`, or `forget()` method) all transaction IDs returned by `recover()`.

- Report heuristic completions

  If a resource manager reports a heuristic exception, the Transaction Recovery Service records the heuristic exception in the server log and calls `forget()` if the `Forget Heuristics` configuration attribute is enabled. If the `Forget Heuristics` configuration attribute is not enabled, refer to your database vendor's documentation for information about resolving heuristic completions. See Handling Heuristic Completions .

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Prior to WebLogic Server 12.2.1.3.0, when a WebLogic JMS resource was enlisted in a subordinate domain in a transaction that spanned domains, and if the WebLogic JMS resource later migrated to a different server before the completion of the transaction, then the automatic transaction recovery of the migrated WebLogic JMS resource and transactions would occur only after the WebLogic JMS resource was migrated back to its original location. This recovery is now supported, however, this recovery does not provide continuity of existing inflight transactions. In case of migration due to server crash, or if WebLogic JMS did not shutdown gracefully then there is a possibility of some exceptions occurring and leading to transactions rolling back.

The Transaction Recovery Service provides the following benefits:

- Maintains consistency across resources

  The Transaction Recovery Service handles transaction recovery in a consistent, predictable manner: For a transaction for which a commit decision has been made but is not yet committed before a crash, and `XAResource.recover()` returns the transaction ID, the Transaction Recovery Service consistently calls `XAResource.commit()`; for a transaction for which a commit decision has not been made before a crash, and `XAResource.recover()` returns its transaction ID, the Transaction Recovery Service consistently calls `XAResource.rollback()`. With consistent, predictable transaction recovery, a transaction manager crash by itself cannot cause a mixed heuristic completion where some branches are committed and some are rolled back.
- Persists in achieving transaction resolution

  If a resource manager crashes, the Transaction Recovery Service must eventually call `commit()` or `rollback()` for each prepared transaction until it gets a successful return from `commit()` or `rollback()`. The attempts to resolve the transaction can be limited by setting the `AbandonTimeoutSeconds` configuration attribute. See Abandoning Transactions .

Recovering Transactions For a Failed Non-Clustered Server

To recover transactions for a failed server, follow these steps:

1. Move (or make available) the persistent store DAT file (which contains all transaction log records) from the failed server to a new server.
2. Set the path for the default persistent store with the path to the data file. See Setting the Path for the Default Persistent Store .
3. Start the new server. The Transaction Recovery Service searches all transaction log files for incomplete transactions and completes them as described in Transaction Recovery Service Actions After a Crash.

When moving transaction log records after a server failure, make all transaction log records available on the new machine before starting the server there. Otherwise, transactions in the process of being committed at the time of a crash might not be resolved correctly, resulting in application data inconsistencies. Accomplish this by storing persistent store data files on a dual-ported disk available to both machines. As in the case of a planned migration, update the default file store `directory` attribute with the new path before starting the server if the pathname is different on the new machine.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> The Transaction Recovery Service is designed to gracefully handle transaction recovery after a crash. Oracle recommends that you attempt to restart a crashed server and allow the Transaction Recovery Service to handle incomplete transactions, rather than move the server to a new machine.

Recovering Transactions For a Failed Clustered Server

When a clustered server fails, you have the following options for recovering transactions:

- Server Migration
- Automatic Transaction Recovery Service Migration
- Manual Transaction Recovery Service Migration
- Managed Server Independence
- Limitations of Migrating the Transaction Recovery Service
- Preparing to Migrate the Transaction Recovery Service
- Constraining Servers to Which the Transaction Recovery Service Can Migrate
- Manually Migrating the Transaction Recovery Service to the Original Server

Server Migration

For clustered servers, WebLogic Server enables you to migrate a failing server to a new machine, including the Transaction Recovery Service. When the server migrates to another machine, it must be able to locate the transaction log records to complete or recover transactions. Transaction log records are stored in the default persistent store for the server. If you plan to migrate clustered servers in the event of a failure, you must set up the default persistent store so that it stores records in a shared storage system that is accessible to any potential machine to which a failed migratable server might be migrated. For highest reliability, use a shared storage solution that is itself highly available—for example, a storage area network (SAN).

For information about server migration, see Whole Server Migration in *Administering Clusters for Oracle WebLogic Server*.

For more information about setting default persistent store options, see:

- Setting the Path for the Default Persistent Store
- Setting the Default Persistent Store Synchronous Write Policy

Automatic Transaction Recovery Service Migration

You can specify to have the Transaction Recovery Service automatically migrated from an unhealthy server instance to a healthy server instance, with the help of the server health monitoring services. This way the backup server can complete transaction work for the failed server. See Roadmap for Configuring Automatic Migration of the JTA Transaction Recovery Service in *Administering Clusters for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Manual Transaction Recovery Service Migration

When a clustered server crashes, you can manually migrate the Transaction Recovery Service from the crashed server to another server in the same cluster using the WebLogic Remote Console or the command-line interface.

You can also configure WebLogic Server to automatically migrate the Transaction Recovery Service to a healthy candidate server based with the help of WebLogic Server health monitoring of singleton services. See Automatic Transaction Recovery Service Migration.

- What Occurs During Transaction Recovery Service Migration

What Occurs During Transaction Recovery Service Migration

When manual or automatic service migration takes place, the following events occur:

1. The Transaction Recovery Service on the backup server takes ownership of the transaction log from the crashed server.
2. The Transaction Recovery Service searches all transaction log records from the failed server for incomplete transactions and completes them as described in Transaction Recovery Service Actions After a Crash.
3. If the Transaction Recovery Service on the backup server successfully completes all incomplete transactions from the failed server, the server releases ownership of the Transaction Recovery Service for the failed server so the failed server can reclaim it upon restart.

A server can perform transaction recovery for multiple failed servers. While recovering transactions for other servers, the backup server continues to process and recover its own transactions. If the backup server fails during recovery, you can migrate the Transaction Recovery Service to yet another server, which continues the transaction recovery. You can also manually migrate the Transaction Recovery Service back to the original failed server using the WebLogic Remote Console or the command-line interface. See Manually Migrating the Transaction Recovery Service to the Original Server.

When a backup server completes transaction recovery for a server, it releases ownership of the Transaction Recovery Service for the failed server. When you restart a failed server, it attempts to reclaim ownership of its Transaction Recovery Service. If a backup server is in the process of recovering transactions when you restart the failed server, the backup server stops recovering transactions, performs some internal cleanup, and releases ownership of the Transaction Recovery service so the failed server can reclaim it and start properly. The failed server then completes its own transaction recovery.

If a backup server still owns the Transaction Recovery Service for a failed server and the backup server is inactive when you attempt to restart the failed server, the failed server does not start because the backup server cannot release ownership of the Transaction Recovery Service. This is also true if the fail back mechanism fails or if the backup server cannot communicate with the Administration Server. You can manually migrate the Transaction Recovery using the WebLogic Remote Console or the command-line interface.

Managed Server Independence

Prior to WebLogic Server 10.0, when a cluster's primary Managed Server was booted, but was unable to contact the Administration Server (mostly because that Administration Server had not started yet), then the primary Managed Server would automatically go into MSI (managed server independence) mode and continue to boot up using its local configuration information. During a manual migration of the Transaction Recovery Service, this situation posed a

Copyright © 2007, 2026, Oracle and/or its affiliates.

potential risk that a backup server was still recovering TLog data on behalf of the primary Managed Server, which could then lead to concurrent access to TLog and potential corruption of the TLog.

To avoid risking potential TLog corruption, there is a `strictOwnershipCheck` property on the JTAMigratableTargetMBean. This way, when a primary Managed Server attempts to boot up and it finds that it cannot connect to the Administration Server (for the manual JTA migration policy) or the Singleton Master (for the automatic JTA migration policy), then it verifies its independence by checking the value of the `strictOwnershipCheck`, as follows:

- `True` – This is the recommended setting. The primary Managed Server throws an exception and fail to boot.
- `False` – The primary Managed Server skips the Transaction Recovery Service failback, then it can boot successfully. This poses the same TLog corruption risk as in WebLogic Server 9.2 or earlier.

Limitations of Migrating the Transaction Recovery Service

When manually or automatically migrating the Transaction Recovery Service, the following limitations apply:

- You cannot migrate the Transaction Recovery Service to a backup server from a server that is running. You must stop the server before migrating the Transactions Recovery Service.
- The backup server does not accept new transaction work for the failed server. It only processes incomplete transactions.
- The backup server does not process heuristic log files.
- The backup server only processes log records written by WebLogic Server. It does not process log records written by gateway implementations, including WebLogic Tuxedo Connector.

In addition to the limitations described above, the following rules also apply when WebLogic Server 10.0 or later is configured to automatically migrate the Transaction Recovery Service:

- If the cluster also contains servers from earlier releases of WebLogic Server, the primary server and backup servers must be WebLogic Server 10.0 or later. To enforce this when automatic migration is enabled, on the WebLogic Remote Console, only WebLogic Server 10.0 or later servers appear in the Candidate Servers **Available** list.
- Manual service migration is supported between release 9.2 or earlier servers and release 10.0 or later servers if no migration scripts are used.

Preparing to Migrate the Transaction Recovery Service

To migrate the Transaction Recovery Service from a failed server in a cluster to another server (backup server) in the same cluster, the backup server must have access to the transaction log records from the failed server. Therefore, you must store default persistent store data files on persistent storage available to all potential backup servers in the cluster. Consider the following:

- A storage area network (SAN) is recommended.
- If you use an NFS file system, configure the NFS server so that disk writes are not cached.
- Regardless of the storage solution, test failover and migration.

Copyright © 2007, 2026, Oracle and/or its affiliates.

With some storage solutions, particularly some versions of NFS, file locking can be an issue. You may need to configure your storage server to handle file locks effectively or you may need to turn off file locking for the WebLogic file store.

> **Note**
>
> NFS storage may not fully protect transactional data, as it may be configured to silently buffer synchronous write requests in volatile memory. If a file store Directory is located on an NFS mount, and the file store's Synchronous Write Policy is anything other than `Disabled`, check your NFS implementation and configuration to make sure that it is configured to support synchronous writes. A `Disabled` synchronous write policy does not perform synchronous writes, but, as a consequence, is generally not transactionally safe. You may detect undesirable buffering of synchronous write requests by observing high persistent message or transaction throughput that exceeds the physical capabilities of your storage device. On the NFS server, check the synchronous write setting of the exported NFS directory hosting your File Store. A SAN based file store, or a JDBC store, may provide an easier solution for safe centralized storage. See File Locking and NFS in *Tuning Performance of Oracle WebLogic Server*.

The following persistent store rules apply when manually or automatically migrating the Transaction Recovery Service:

- The default persistent store cannot be shared by JTA and other migratable services. Other migratable services, such as JMS services, must use another custom store if they are targeted to a migratable target.
- If post-deactivation and pre-activation scripts are specified to perform any dismounting and mounting of the default store, then the Node Manager must be configure and running on all candidate machines.

The Administration Server must be available when the primary server starts up, fails over, or fails back. This is required to guarantee that the Transaction Recovery Service gets exclusive ownership to its TLog correctly and without conflict. When the primary server starts up, the Transaction Recovery Service connects to Administration Server to get the latest information about JTA. And should failover/failback occur, the Transaction Recovery Service saves the latest information to Administration Server.

When migrating the Transaction Recovery Service from a server, you must stop the failing or failed server before actually migrating the Transaction Recovery Service. If the original server is still running, you cannot migrate the Transaction Recovery Service from it.

All servers that participate in the migration must have a listen address specified in their configuration.

Constraining Servers to Which the Transaction Recovery Service Can Migrate

You may want to limit the choices of the servers to use as a Transaction Recovery Service backup for a server in a cluster. For example, all servers in your cluster may not have access to the transaction log records for a server.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> You must include the original server in the list of chosen servers so that you can manually migrate the Transaction Recovery Service back to the original server, if need be. The WebLogic Remote Console enforces this rule.

Manually Migrating the Transaction Recovery Service to the Original Server

After completing transaction recovery for a failed server, a backup server releases ownership of the Transaction Recovery Service so that the original server can reclaim it when the server is restarted. If the backup server stops (crashes) for any reason before it completes transaction recovery, the original server cannot reclaim ownership of the Transaction Recovery Service and does not start.

You can manually migrate the Transaction Recovery Service back to the original server by selecting the original server as the destination server. The backup server must not be running when you migrate the service back to the original server. Follow the instructions below.

> **Note**
>
> Please note the following:
>
> - If a backup server fails before completing the transaction recovery actions, the primary server cannot reclaim ownership of the Transaction Recovery Service and recovery is not re-attempted after the rebooting server. Therefore, you must attempt to manually re-migrate the Transaction Recovery Service to another backup server.
> - If you restart the original server while the backup server is recovering transactions, the backup server gracefully releases ownership of the Transaction Recovery Service. You do not need to stop the backup server. See Recovering Transactions For a Failed Clustered Server.

How to Remove Transaction Records

Before deleting TLogs the WebLogic Server instance should be shutdown gracefully to allow the completion of as many transactions as possible.

> **Note**
>
> You should delete TLogs only in an extreme case. Deleting the TLogs removes transaction records, resulting in heuristic failures. For example, see Moving a Server.

The location of TLogs is dependent whether you are using the default store, a JDBC TLog store, and if LLR is a participating resource in a transaction.

- When one resource involved in the transaction is a LLR, then the TLogs are stored in two locations.
  - The transaction records are stored in a database table. See How to Remove the TLog in the LLR Database.

Copyright © 2007, 2026, Oracle and/or its affiliates.

  - The server and resource checkpoints are stored in the default store or a JDBC TLog store. See How to Remove the TLog Files from the Default Store and How to Remove the TLog from a JDBC TLog Store.

- If there are no participating LLR in the transactions, the transaction records, server checkpoints, and resource checkpoints are all saved to the TLog file in the default store or a JDBC TLog store. See How to Remove the TLog Files from the Default Store and How to Remove the TLog from a JDBC TLog Store.
- How to Remove the TLog in the LLR Database
- How to Remove the TLog Files from the Default Store
- How to Remove the TLog from a JDBC TLog Store

How to Remove the TLog in the LLR Database

Default name of the LLR table is `WL_LLR_`*`SERVERNAME`*, where *`SERVERNAME`* is the name of the server instance. To delete the LLR TLog that is kept in the database, remove all the records from the table by issuing `drop table WL_LLR_`*`SERVERNAME`*.

How to Remove the TLog Files from the Default Store

To remove the TLogs in a the default store, delete all files having the following pattern:

`$`*`DOMAIN_HOME`*`/servers/`*`servername`*`/data/store/default/_WLS_`*`SERVERNAME`*`xxxxxx.DAT`

where xxxxxx are integers ranging from 0 to 9.

> **Note**
>
> If the default store contains a configured JMS file store, deleting the TLog also deletes the JMS File Store. In this case, before deleting the TLog files, first export the JMS messages to another location. You can then safely delete the TLog files and import the JMS messages back to the original store. See Managing JMS Messages in *Administering JMS Resources for Oracle WebLogic Server*.

How to Remove the TLog from a JDBC TLog Store

The name of the JDBC TLog store is the PrefixName prepended to the name of the server hosting the JDBC TLog store and ends in "_". For example, a valid JDBC TLog store name using the default Prefix Name is `TLOG_MyServer_` where `TLOG_` is the Prefix Name and `MyServer` is the name of the server hosting the JDBC TLog store. You database administrator can remove the existing TLog information from your JDBC TLog store.

Copyright © 2007, 2026, Oracle and/or its affiliates.

4

Using Transaction Guard

Transaction Guard provides a generic infrastructure for applications to use for at-most-once execution during planned and unplanned outages and duplicate submissions. Applications use the logical transaction ID to determine the outcome of the last transaction open in a database session following an outage.

Without Transaction Guard, applications that attempt to replay operations following outages can cause logical corruption by committing duplicate transactions.

Transaction Guard provides these benefits:

- Preserves the commit outcome
- Ensures a known outcome for every transaction
- Provides at-most-once transaction execution

This chapter contains the following topics:

- Overview of Transaction Guard
  The main purpose of Transaction guard is to provide *at-most execution semantics* in case of failures. The semantics assure that end-user transactions are executed on time and at-most-once.
- Enabling Transaction Guard
  Transaction Guard must be configured both in the data source and in the database service.
- Data Source Statistics for Transaction Guard
  Learn about Data source statistics available on the weblogic.management.runtime.JDBCDataSourceRuntimeMBean for transaction guard.

Overview of Transaction Guard

The main purpose of Transaction guard is to provide *at-most execution semantics* in case of failures. The semantics assure that end-user transactions are executed on time and at-most-once.

For current applications, in case of communication failure to the server, determining the outcome of the last commit operation in a guaranteed and scalable manner becomes a challenge. In many cases, the end users are asked to follow certain steps to avoid resubmitting duplicate request. For example, some applications warn users not to click the `Submit` button twice because if it is not followed, then users may unintentionally purchase the same items twice and submit multiple payments for the same invoice.

Without Transaction Guard, if a transaction has been started and commit has been issued, the commit message that is sent back to the client is not durable. The client is left not knowing whether the transaction committed. The transaction cannot be validly resubmitted if the non-transactional state is incorrect or if it already committed. In the absence of guaranteed commit and completion information, re-submission can lead to transactions applied more than once and in a session with the incorrect state.

Copyright © 2007, 2026, Oracle and/or its affiliates.

It's possible to use global XA transactions to limit the problem described above. However, there is overhead associated with the protocols required for XA processing. Further, some features like application continuity are not available with XA transactions.

WebLogic Server Transaction Management integrates with Transaction Guard internally to determine the state (commit outcome) of transactions during failures. The benefits of this integration is reducing heuristic exceptions in the case of global transactions with more than one resource, and reducing system exceptions in the case of one-phase transactions with only one resource. The implementation of this integration is internal to WebLogic Server and does not require changes to the application code.

If there is an exception during the completion of the transaction, WebLogic Server uses Transaction Guard to determine the transaction result. No changes are needed in the application code.

For more information about Transaction Guard, see Transactions in the Oracle Database guide.

Enabling Transaction Guard

Transaction Guard must be configured both in the data source and in the database service.

Topics

- Enabling Transaction Guard for WebLogic Data Sources
- Enabling Transaction Guard for Database Service

Enabling Transaction Guard for WebLogic Data Sources

Except TwoPhaseCommit, transaction guard supports all non-XA Oracle data source driver types and all WebLogic data source transaction types like EmulateTwoPhaseCommit, OnePhaseCommit, LoggingLastResource, and None.Enable transaction guard support on a data source by setting the connection property `weblogic.jdbc.commitOutcomeEnabled` to `true`.
For example, the following is a sample from a WLST script that creates a data source:

```
jdbcSR = create(dsname, 'JDBCSystemResource')
jdbcResource = jdbcSR.getJDBCResource()
driverParams = jdbcResource.getJDBCDriverParams()
driverProperties = driverParams.getProperties()
tgprop = driverProperties.createProperty('weblogic.jdbc.commitOutcomeEnabled')
tgprop.setValue('true')
```

While trying to get the status of the transaction using Transaction Guard, it is possible for the operation to time out and the transaction will fail. The timeout for connections on a datasource with transaction type EmulateTwoPhaseCommit, OnePhaseCommit, or LoggingLastResource uses the XA completion-timeout value. The timeout for connections on a datasource with transaction type None (e.g., outside a transaction) is 10 seconds.

To override the default value, use the `weblogic.jdbc.commitOutcomeRetryMaxSeconds` connection property. For example, add the following to the WLST script above.

```
tgseconds = driverProperties.createProperty(
    'weblogic.jdbc.commitOutcomeRetryMaxSeconds')
tgseconds.setValue('60')
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> This feature is supported in a global transaction that has both XA and non-XA resources or has multiple EmulateTwoPhaseCommit resources. Global transactions with more than one participating resource can still experience heuristic outcomes. The use of the Transaction Guard feature on the datasource(s) does not guard against the inherent potential for non-atomic/heuristic outcome for this type of global transaction.

Enabling Transaction Guard for Database Service

The database service that is specified in the URL for the data source must be configured with `the-commit_outcome` parameter set to `TRUE` and optionally set the retention timeout.

For example:

```
srvctl modify service -d mydb -s myservice -e TRANSACTION -commit_outcome TRUE -
retention 604800 -rlbgoal SERVICE_TIME -clbgoal SHORT
```

See Database Configuration for Transaction Guard

Data Source Statistics for Transaction Guard

Learn about Data source statistics available on the weblogic.management.runtime.JDBCDataSourceRuntimeMBean for transaction guard.

The following related statistics are available on the weblogic.management.runtime.JDBCDataSourceRuntimeMBean.

public long getResolvedAsCommittedTotalCount();

The cumulative total number of commit outcomes successfully resolved as committed in this data source since the data source was deployed.

public long getResolvedAsNotCommittedTotalCount();

The cumulative total number of commit outcomes successfully resolved as not committed in this data source since the data source was deployed.

public long getUnresolvedTotalCount();

The cumulative total number of commit outcomes unsuccessfully resolved in this data source since the data source was deployed.

public long getCommitOutcomeRetryTotalCount();

The cumulative total number of commit outcome query retries conducted before resolving the outcome or exceeding the retry seconds in this data source since the data source was deployed.

See JDBCDataSourceRuntimeMBean.

Copyright © 2007, 2026, Oracle and/or its affiliates.

5

Transaction Service

Learn how to write information for transactional applications to be used with WebLogic Server. Also, understand the capabilities and limitations of the transaction service and how to use it with EJBs, RMI applications, and the Object Transaction Service (OTS).

- About the Transaction Service
  WebLogic Server provides a Transaction Service that supports transactions in EJB and RMI applications.
- Capabilities and Limitations
  Learn about the capabilities and limitations of the Transaction Service that supports EJB and RMI applications.
- Transaction Scope
  The scope of a transaction refers to the environment in which the transaction is performed. WebLogic Server supports transactions on standalone servers, between non-clustered servers, between clustered servers within a domain, and between domains.
- Transaction Service in EJB Applications
  The WebLogic Server EJB container provides a Transaction Service that supports the two types of transactions in WebLogic Server EJB applications such as container-managed transactions and bean-managed transactions.
- Transaction Service in RMI Applications
  WebLogic Server provides a Transaction Service that supports transactions in WebLogic Server RMI applications. In RMI applications, the client or server application makes explicit method invocations on the `UserTransaction` object to begin, commit, and roll back transactions.
- Transaction Service Interoperating with OTS
  WebLogic Server provides a Transaction Service that supports interoperation with the Object Transaction Service (OTS).

About the Transaction Service

WebLogic Server provides a Transaction Service that supports transactions in EJB and RMI applications.

In the WebLogic Server EJB container, the Transaction Service provides an implementation of the transaction services described in the Jakarta Enterprise Beans Specification 4.0 at `https://jakarta.ee/specifications/enterprise-beans/4.0/`.

For EJB and RMI applications, WebLogic Server also provides the `jakarta.transaction` and `javax.transaction.xa` packages, which implement the Java Transaction API (JTA) for Java applications. For more information about JTA, see the Java Transaction API (JTA) Specification 2.0, published at `https://jakarta.ee/specifications/transactions/2.0/`. For more information about the `UserTransaction` object that applications use to demarcate transaction boundaries, see `weblogic.transaction.UserTransaction` in the *Java API Reference for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Capabilities and Limitations

Learn about the capabilities and limitations of the Transaction Service that supports EJB and RMI applications.

- Lightweight Clients with Delegated Commit
- Client-initiated Transactions
- Transaction Integrity
- Transaction Termination
- Flat Transactions
- Relationship of the Transaction Service to Transaction Processing
- Multithreaded Transaction Client Support
- Transaction Id
- Transaction Name and Properties
- Transaction Status
- Transaction Statistics
- General Constraints

Lightweight Clients with Delegated Commit

A lightweight client runs on a single-user, unmanaged desktop system that has irregular availability. Owners may turn their desktop systems off when they are not in use. These single-user, unmanaged desktop systems should not be required to perform network functions such as transaction coordination. In particular, unmanaged systems should not be responsible for ensuring atomicity, consistency, isolation, and durability (ACID) properties across failures for transactions involving server resources. WebLogic Server remote clients are lightweight clients.

The Transaction Service allows lightweight clients to do a delegated commit, which means that the Transaction Service allows lightweight clients to begin and terminate transactions while the responsibility for transaction coordination is delegated to a transaction manager running on a server machine. Client applications do not require a local transaction server. The remote implementation of `UserTransaction` that EJB or RMI clients use delegates the actual responsibility of transaction coordination to the transaction manager on the server.

Client-initiated Transactions

A client, such as an applet, can obtain a reference to the `UserTransaction` and `TransactionManager` objects using JNDI. A client can begin a transaction using either object reference. To get the `Transaction` object for the current thread, the client program must invoke the `((TransactionManager)tm).getTransaction()` method.

Transaction Integrity

Checked transaction behavior provides transaction integrity by guaranteeing that a `commit` does not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. The Transaction Service *provides* checked

Copyright © 2007, 2026, Oracle and/or its affiliates.

transaction behavior that is equivalent to that provided by the request/response inter-process communication models defined by The Open Group.

Transaction Termination

WebLogic Server allows transactions to be terminated *only* by the client that created the transaction.

> **Note**
>
> The client may be a server object that requests the services of another object.

Flat Transactions

WebLogic Server implements the flat transaction model. Nested transactions are *not* supported.

Relationship of the Transaction Service to Transaction Processing

The Transaction Service relates to various transaction processing servers, interfaces, protocols, and standards in the following ways:

- **Support for The Open Group XA interface.**The Open Group Resource Managers are resource managers that can be involved in a distributed transaction by allowing their two-phase commit protocol to be controlled using The Open Group XA interface. WebLogic Server supports interaction with The Open Group Resource Managers.
- **Support for the OSI TP protocol.** Open Systems Interconnect Transaction Processing (OSI TP) is the transactional protocol defined by the International Organization for Standardization (ISO). WebLogic Server *does not* support interactions with OSI TP transactions.
- **Support for the LU 6.2 protocol.** Systems Network Architecture (SNA) LU 6.2 is a transactional protocol defined by IBM. WebLogic Server *does not* support interactions with LU 6.2 transactions.
- **Support for the ODMG standard.** ODMG-93 is a standard defined by the Object Database Management Group (ODMG) that describes a portable interface to access Object Database Management Systems. WebLogic Server *does not* support interactions with ODMG transactions.

Multithreaded Transaction Client Support

WebLogic Server supports multithreaded transactional clients. Clients can make transaction requests concurrently in multiple threads.

Transaction Id

The Transaction Service assigns a transaction identifier (`XID`) to each transaction. This ID can isolate information about a specific transaction in a log file. You can retrieve the transaction identifier using the `getXID` method in the `weblogic.transaction.Transaction` interface. For detailed information on methods for getting the transaction identifier, see `weblogic.transaction.Transaction` in the *Java API Reference for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Transaction Name and Properties

WebLogic JTA provides extensions to `jakarta.transaction.Transaction` that support transaction naming and user-defined properties. These extensions are included in the `weblogic.transaction.Transaction` interface.

The transaction name indicates a type of transaction (for example, funds transfer or ticket purchase) and should not be confused with the transaction ID, which identifies a unique transaction on a server. The transaction name makes it easier to identify a transaction type in the context of an exception or a log file.

User-defined properties are key/value pairs, where the key is a string identifying the property and the value is the current value assigned to the property. Transaction property values must be objects that implement the `Serializable` interface. You manage properties in your application using the set and get methods defined in the `weblogic.transaction.Transaction` interface. Once set, properties stay with a transaction during its entire lifetime and are passed between machines as the transaction travels through the system. Properties are saved in the transaction log, and are restored during crash recovery processing. If a transaction property is set more than once, the latest value is retained.

For detailed information on methods for setting and getting the transaction name and transaction properties, see `weblogic.transaction.Transaction` in the *Java API Reference for Oracle WebLogic Server*.

Transaction Status

The Java Transaction API provides transaction status codes using the `jakarta.transaction.Status` class. Use the `getStatusAsString` method in `weblogic.transaction.Transaction` to return the status of the transaction as a string. The string contains the major state as specified in `jakarta.transaction.Status` with an additional minor state (such as `logging` or `pre-preparing`).

Transaction Statistics

Transaction statistics are provided for all transactions handled by the transaction manager on a server. These statistics include the number of total transactions, transactions with a specific outcome (such as committed, rolled back, or heuristic completion), rolled back transactions by reason, and the total time that transactions were active. For detailed information on transaction statistics, see Monitoring Transactions.

General Constraints

The following constraints apply to the Transaction Service:

- In WebLogic Server, a client or a server object *cannot* invoke methods on an object that is infected with (or participating in) another transaction. The method invocation issued by the client or the server instance returns an exception.
- In WebLogic Server, clients using third-party implementations of the Java Transaction API (for Java applications) *are not* supported.
- The transaction log buffer is limited to 250 KB. If your application includes very large transactions that require transaction log writes that exceed this value, WebLogic Server instance throws an exception. In that case, you must reconfigure your application to work around the buffer size.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Transaction Scope

The scope of a transaction refers to the environment in which the transaction is performed. WebLogic Server supports transactions on standalone servers, between non-clustered servers, between clustered servers within a domain, and between domains.

To enable inter-domain transaction support, see Configuring Secure Inter-Domain and Intra-Domain Transaction Communication.

Transaction Service in EJB Applications

The WebLogic Server EJB container provides a Transaction Service that supports the two types of transactions in WebLogic Server EJB applications such as container-managed transactions and bean-managed transactions.

- **Container-managed transactions**. In container-managed transactions, the WebLogic Server EJB container manages the transaction demarcation. Transaction attributes in the EJB deployment descriptor determine how the WebLogic Server EJB container handles transactions with each method invocation.
- **Bean-managed transactions**. In bean-managed transactions, the EJB manages the transaction demarcation. The EJB makes explicit method invocations on the `UserTransaction` object to begin, commit, and roll back transactions. For more information about `UserTransaction` methods, see `weblogic.transaction.UserTransaction` in the *Java API Reference for Oracle WebLogic Server*.

For an introduction to transaction management in EJB applications, see Transactions in WebLogic Server EJB Applications and Transactions Sample EJB Code .

Transaction Service in RMI Applications

WebLogic Server provides a Transaction Service that supports transactions in WebLogic Server RMI applications. In RMI applications, the client or server application makes explicit method invocations on the `UserTransaction` object to begin, commit, and roll back transactions.

For more information about `UserTransaction` methods, see the online javadoc. For an introduction to transaction management in RMI applications, see Transactions in WebLogic Server RMI Applications and Transactions Sample RMI Code.

Transaction Service Interoperating with OTS

WebLogic Server provides a Transaction Service that supports interoperation with the Object Transaction Service (OTS).

For this release, WebLogic Server interoperates with OTS in the following scenarios:

- Server-Server 2PC
- Client Demarcated Transactions

Server-Server 2PC

In this situation, a server-to-server 2PC transaction is completed using interposition. The originating server creates an Xid and propagates the transaction to the target server. The

Copyright © 2007, 2026, Oracle and/or its affiliates.

target server registers itself as a resource with the originating server. The originating server drives the completion of the transaction. Logging Last Resource (LLR) transaction optimization, as described in Logging Last Resource Transaction Optimization, is not supported.

Client Demarcated Transactions

The client starts a transaction on the server using the OTS client APIs. The client then retrieves the Xid from this transaction and then propagates this per-request until the transaction is committed. Although the client initiates the transaction, all the commit processing occurs on the server.

Copyright © 2007, 2026, Oracle and/or its affiliates.

6

Transaction Recovery Spanning Multiple Sites or Data Centers

Learn about the best practices for XA transaction recovery of WebLogic domains across physical sites as part of a Disaster Recovery (DR) solution.

- Understanding XA Transaction Recovery in Disaster Recovery
  Maximizing availability and providing protection from unforeseen disasters and natural calamities are key requirements of a disaster recovery solution for an enterprise deployment. One aspect of a disaster recovery solution is to ensure that all XA transactions of affected WebLogic domains can be recovered if a production site is no longer available.
- Active-Passive XA Transaction Recovery
  Learn about the domain configuration and requirements for XA transactions in an active-passive recovery solution.
- Active-Active Stretch Cluster XA Transaction Recovery
- Additional Information on Maximum Availability Architecture
  Oracle provides a number of resources which provide additional information on how to configure environments that maximize availability.

Understanding XA Transaction Recovery in Disaster Recovery

Maximizing availability and providing protection from unforeseen disasters and natural calamities are key requirements of a disaster recovery solution for an enterprise deployment. One aspect of a disaster recovery solution is to ensure that all XA transactions of affected WebLogic domains can be recovered if a production site is no longer available.

For transaction recovery, the following solutions are provided:

- **Active-Passive**: These solutions involve setting up and pairing a standby site at a geographically different location with an active (production) site. The standby site is normally in a passive mode; it is started when the production site is not available. See Active-Passive XA Transaction Recovery.
- **Active-Active Stretch Cluster**: In this architecture, a cluster stretches across two sites. Transaction recovery across sites takes place by service or server migration. See Active-Active Stretch Cluster XA Transaction Recovery.

> **Note**
>
> Transactions that enlist a WebLogic Server JMS resource cannot be recovered in active-active recovery solutions.

See Overview of Disaster Recovery in *Oracle® Fusion Middleware Disaster Recovery Guide* for more information on disaster recovery solutions for enterprise deployments. The following sections provide the requirements and guidelines to recover XA transactions from failed production domains.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Active-Passive XA Transaction Recovery

Learn about the domain configuration and requirements for XA transactions in an active-passive recovery solution.

Requirements for XA Transactions in an Active-Passive Disaster Recovery Solution

This section provides the conditions and configuration requirements necessary to enable the successful recovery of XA transactions following the failure of a production site in an active-passive disaster recovery solution:

- All active-passive domain pairs are configured with symmetric topology, they are identical and have the same domain configurations.
- With WebLogic Continuous Availability failover can be orchestrated with Oracle Site Guard. Oracle Enterprise Manager Cloud Control is one option that provides the ability to manage disaster recover of WebLogic Server domains across multiple data centers.
- The ability to maintain the workload at less than full capacity on the active and passive site during runtime in order to achieve a consistent capacity during runtime and recovery.
- Only hostnames (not static IPs) must be used to specify the listen address of managed servers. Configuration of these hostnames must be identical on all sites. As hostnames between sites are identical but IPs are not, the hostname provides the dynamic ability to simply start an identically configured server or domain on the recovery site.
  - Before initiating the Transaction Recovery service by starting servers in the passive domain, update the DNS server to point the DNS names to the machine(s) in passive data center.

    For example: Active domain Domain1 has two managed servers running on two machines Mc1 and Mc2. In domain configuration, use the corresponding DNS names dns-1 and dns-2. When the active domain fails and we want to activate corresponding passive domain, update the DNS server and change configuration to point dns-1 and dns-2 to Mc3 and Mc4 respectively. Then start passive Domain2.
  - Do not use DNS names that include an underscore, they are not valid in WebLogic Server domains. DNS names with a dash are valid.
- You have several options to store the TLog: a default store, JDBC TLog, LLR, and a determiner resource. A default store must be in a common area (usually NFS or SAN). A JDBC TLog uses a database as a common storage location to all WLS servers and is typically replicated using DataGuard or Active DataGuard to ensure high availability. The determiner-based NoTLog feature is deprecated as of Oracle WebLogic Server 26.1.0.0.0; see XA Transactions without Transaction TLog Write.
- Transaction service migration within a cluster is only supported if the entire cluster, including the corresponding domains and servers, is failed over to a recovery site. Specifically, the administrator must insure the node manager and entire cluster, corresponding domains, and any impacted servers have been shutdown on failed site before starting them all on the recovery site.
- Transactions that span WebLogic domains can only be recovered in a site failover if all domains involved in the transaction are failed over.
- The domain information is kept in a shared location(s) to ensure domain configurations are in sync. Applications are kept in a shared location(s) to ensure they are in sync.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Pack/Unpack could be another approach to keep configurations in sync.

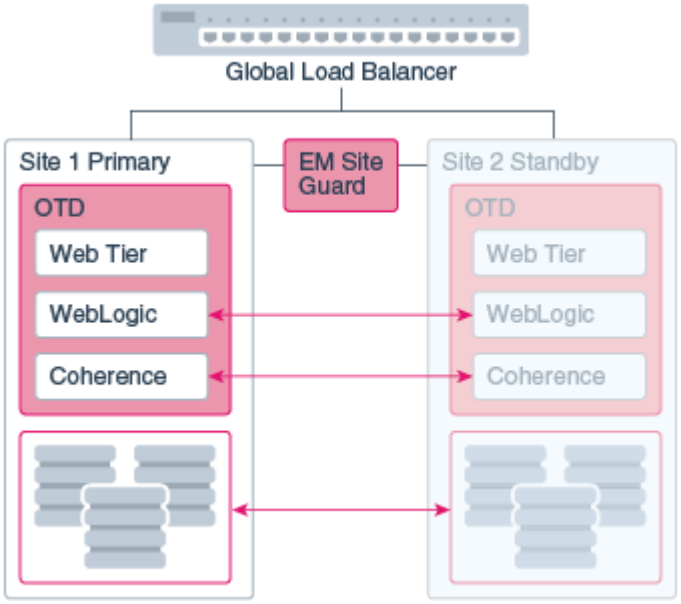
See Setting Up and Managing Disaster Recovery Sites in Oracle® Fusion Middleware Disaster Recovery Guide for detailed information on conditions and requirements for setting up active and passive recovery sites for enterprise deployments.

- Example Active-Passive Domain Configuration for XA Transaction Recovery

Example Active-Passive Domain Configuration for XA Transaction Recovery

All active-passive domain pairs are configured with symmetric topology, they are identical and have the same domain configurations. The failover process in an Active-Passive architecture is either manual or controlled by an external tool.

Figure 6-1 Domain Configuration for Active-Passive Recovery



An application running on Site1 starts a transaction. After it calls commit, the entire application infrastructure tier comes down. In this example, the application session replication, file system replication, and DB replication are taking place between the two sites.

If the entire WebLogic Server tier has not come down then all servers need to be shutdown and all the identical servers in the passive domain need to be started. Since the domains, clusters, servers, and resource all have identical names, as soon as the servers are started, recovery will commence and all transactions will recover. It is recommended to do a graceful shutdown of any server to allow work to drain.

Active-Active Stretch Cluster XA Transaction Recovery

Learn about the requirements and domain configuration for XA transactions in an active-active recovery solution, where a WebLogic Server Cluster stretches across two sites.
When a server of the cluster fails, server and resource checkpoints are still written to the TLOG. Checkpoints are written when the resources are first involved in a global transaction,

Copyright © 2007, 2026, Oracle and/or its affiliates.

updated only if there are changes to the transaction participants, and purged only if they are no longer used or become unavailable. Because checkpoints are created early in the transaction, as long as there are no changes to the participants in the global transactions, there is little danger of a checkpoint being out of sync during transaction service migration or transaction recovery spanning multiple sites or data centers. another server of the cluster can take over transaction recovery with service or server migration. This kind of architecture requires low latency between the sites. See Multicast and Cluster Configuration in *Administering Clusters for Oracle WebLogic Server*.

This section describes the domain configuration and requirements for XA transactions in an active-active stretch cluster recovery solution.

Requirements for XA Transactions in an Active-Active Stretch Cluster Disaster Recovery Solution

This section describes the conditions and configuration requirements necessary to enable the successful recovery of XA transactions following the failure of a production site in an active-active stretch cluster disaster recovery solution.

Since this architecture uses service or server migration for XA transaction recovery, the requirements are same as the conditions necessary for server migration. See Server Migration.

In addition, the network must meet the following requirements:

- Full support of IP multicast packet propagation. In other words, all routers and other tunneling technologies must be configured to propagate multicast messages to clustered server instances.
- Network latency low enough to ensure that most multicast messages reach their final destination in approximately 10 milliseconds.
- Multicast Time-To-Live (TTL) value for the cluster high enough to ensure that routers do not discard multicast packets before they reach their final destination. For instructions on setting the Multicast TTL parameter, see Configure Multicast Time-To-Live (TTL).

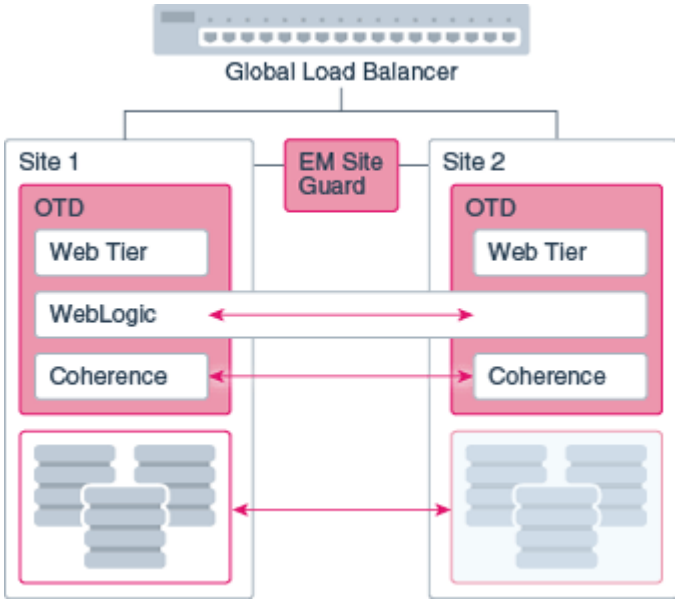
  See If Your Cluster Spans Multiple Subnets In a WAN in *Administering Clusters for Oracle WebLogic Server*.
- Example Active-Active Stretch Cluster for XA Transaction Recovery

Example Active-Active Stretch Cluster for XA Transaction Recovery

In an active-active stretch cluster recovery solution, the JTA service or server migration is used for transaction recovery. This architecture is recommended when there is low latency between sites.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Figure 6-2 Domain Configuration for Active-Active Stretch Cluster Recovery



In this example, an application running on Site 1 starts a transaction. After it calls commit, one or more servers on Site 1 fails. If service or server migration are configured then the surviving servers on Site 2 (in the stretch cluster) will take over recovery for the failed servers.

Additional Information on Maximum Availability Architecture

Oracle provides a number of resources which provide additional information on how to configure environments that maximize availability.

See the following topics:

- http://www.oracle.com/technetwork/database/features/availability/fusion-middleware-maa-155387.html
- *Oracle® Fusion Middleware Disaster Recovery Guide*
- *Oracle® Fusion Middleware High Availability Guide*

Copyright © 2007, 2026, Oracle and/or its affiliates.

7

Configuring Secure Inter-Domain and Intra-Domain Transaction Communication

Learn about secure inter-domain and intra-domain communication and how to configure these secure communication between servers during a transaction.

- What is Secure Inter-Domain and Intra-Domain Transaction Communication?
  In a transaction communication if the participating servers are within the same domain it is an intra-domain communication. When the servers participating in transactions are not in the same domain then it is an inter-domain communication.
- Requirements for Transaction Communication
  There are certain requirements when configuring communication channels for your transaction environment.
- Configuring Secure Channel Communication
  Learn how to configure secure communication channels between servers.

What is Secure Inter-Domain and Intra-Domain Transaction Communication?

In a transaction communication if the participating servers are within the same domain it is an intra-domain communication. When the servers participating in transactions are not in the same domain then it is an inter-domain communication.

For a transaction manager to manage distributed transactions, the transaction manager must be able to communicate with all participating servers and resources to prepare and then commit or rollback the transactions. How a communication channel is configured depends on whether the transaction route is:

- Intra-domain—The transaction communication is between servers participating in transactions within the same domain. You must correctly configure compatible communication channels using either Local Domain Security for all participating servers in global transactions or security interoperability mode.
- Inter-domain—The transaction communication is between servers participating in transactions that are not in the same domain. You must correctly configure compatible communication channels using either Cross Domain Security for all participating domains in global transactions or security interoperability mode.

Communication channels must be secure to prevent a malicious third-party from using man-in-the-middle attacks to affect transaction outcomes and potentially gaining administrative control over one or more domains.

Requirements for Transaction Communication

There are certain requirements when configuring communication channels for your transaction environment.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- The domains and all participating resources must have unique names. That is, you cannot have a server or a domain with the same name as an object in another domain or the domain itself.
- Keep all the domains used by your process symmetric with respect to Cross Domain Security configuration, as settings are set at the domain level.
- Configure one-way SSL to provide additional communication security to protect the transaction from a man-in-the-middle attack.
- Only one data source with *both* of the following attribute conditions participate in a global transaction, regardless of the domain in which the data source is configured:
  - Logging Last Resource or Emulate two-phase Commit is selected.
  - The data source uses a non-XA driver to create database connections.

> **Note**
>
> If more than one LLR or JTS resource participate in a global transaction, naming conflicts will cause the global transaction to fail.

- How to Determine the Communication to Use for Domain Transactions

How to Determine the Communication to Use for Domain Transactions

Use Table 7-1 to determine the usage of Cross Domain Security, Local Domain Security, or Security Interoperability Mode:

Table 7-1 Selecting a Channel Configuration

Channel Configuration	Server Connection	Advantage	Disadvantage
Cross Domain Security	Inter-domain	• Specific users are configured to establish communication between a domain pair. • With SSL, prevents man-in-the-middle attacks.	• More complex configuration. • Any change to the transaction flow, such as changing participants, participant roles (coordinator versus resource or subcoordinator), adding or removing a domain, or changing the transaction route, requires a configuration change.
Local Domain Security	Intra-domain	• A specific user is configured to establish communication between servers in the same domain.	• More complex configuration

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 7-1 (Cont.) Selecting a Channel Configuration

Channel Configuration	Server Connection	Advantage	Disadvantage
Security Interoperability Mode	• Inter-domain • Intra-domain	• Very easy to configure. • No need to understand the transaction flow when configuring Security Interoperability Mode. • When in `default` mode, using the **Admin** channel prevents man-in-the-middle attacks. If **Admin** channel is configured and the Security Interoperability Mode is set to `default`, messages are forwarded using kernel identity. If **Admin** channel is not configured, the `default` Interoperability Mode behaves the same as the `performance` Interoperability Mode, where messages are forwarded using anonymous user.	• All domains that have servers that participate in the global transaction have the same domain trust. This is not as secure as in Cross Domain Security where the trust is established only between a domain pair. • When set to `performance`, you are not required to set domain trust between the domains. • In some configurations, there is a narrow possibility of man-in-the-middle attacks.

Configuring Secure Channel Communication

Learn how to configure secure communication channels between servers.

- Local Domain Security
- Cross Domain Security

Local Domain Security

Local Domain Security uses a credential mapper to enable you to configure compatible communication channels between servers in the same domain in global transactions. You can designate a user to perform all JTA communication between servers in the domain.

When local domain security is enabled, JTA retrieves the designated user's credentials from a credential mapper and uses them to authenticate a subject and then perform the requested operation. The user must be a local user that belongs to the `CrossDomainConnector` user group and exists as an entry in the credential mapper.

See Enable Local Domain Security for JTA in *Oracle WebLogic Remote Console Online Help*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Cross Domain Security

Cross Domain Security uses a credential mapper to enable you to configure compatible communication channels between remote servers in global transactions. For every domain pair that participates in a transaction, a credential mapper is configured. Every domain pair has a different set of credentials which belong to the `CrossDomainConnector` security role.

Although it requires a more complex configuration, Cross Domain Security enables you to tailor trust between individual domains. As described in Cross Domain Security Between WebLogic Server Domains, subsystems such as JMS, JTA, MDB, and WAN replication implement cross-domain security. (The EJB container does not implement the solution for cross-domain security.)

See:

- Enabling Cross Domain Security Between WebLogic Server Domains in *Administering Security for Oracle WebLogic Server*
- Configure a Credential Mapping for Cross-Domain Security in *Administering Security for Oracle WebLogic Server*
- Important Considerations When Configuring Cross Domain Security
- Cross Domain Security is Not Transitive
  Servers participating in a transaction set cross-domain credential mapping with each other.
- Adding Domains to the Exclude List Based on Transaction Participation
  The exclude list provides a mechanism for a server in a domain with Cross Domain Security configured to participate in a transaction with a server in another domain that does not support or have Cross Domain Security enabled.

Important Considerations When Configuring Cross Domain Security

When configuring Cross Domain Security, consider the following guidelines:

- Domain trust is not required for Cross Domain Security.
- For every domain pair that participates in a transaction, a credential mapper must be correctly configured having a set of credentials which belong to the `CrossDomainConnector` security role. If the credential mapping is not correct, transactions across the participating domains fail. See Configure a Credential Mapping for Cross-Domain Security in *Administering Security for Oracle WebLogic Server*.
- To interoperate with WebLogic domains that either do not support Cross Domain Security or have Cross Domain Security disabled, you must add these domains to the `Excluded Domain Names` list of every participating WebLogic Server domain that has Cross Domain Security enabled. If the configuration of the `Excluded Domain Names` list and the `CrossDomainSecurityEnabled` flag is not consistent in all participating domains, branches of the transaction fail.
- If `Cross Domain Security Enabled` flag is enabled and the transaction has participating servers that are remote and local, Cross Domain security will be used for RMI communication with the remote servers.
- When enabling or disabling the `Cross Domain Security Enabled` flag, there may be a period of time where transactions or other remote calls can fail. For transactions, if the commit request fails, the commit is retried after the configuration change is complete. If a transaction RMI call fails during any other request, then the transaction times out and the transaction is rolled back. The rollback is retried until `AbandonTimeoutSeconds`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Cross Domain Security is Not Transitive

Servers participating in a transaction set cross-domain credential mapping with each other.

Unlike domain-trust, the cross domain security configuration is not transitive; that is, because `A` trusts `B` and `B` trusts `C`, it is not therefore also true that `A` trusts `C`.

Consider the follow scenario:

- DomainA has Server1 (coordinator)
- DomainB has Server2 (sub-coordinator)
- DomainC has Server3 and Server4 (Server3 is a sub-coordinator)
- DomainD has Server5 (does not participate in the transaction)

To set the cross-domain credential mapping in this scenario, do the following:

1. Set cross-domain security in DomainA for DomainB
2. Set cross-domain security in DomainB for DomainA
3. Set cross-domain security in DomainA for DomainC
4. Set cross-domain security in DomainC for DomainA
5. Set cross-domain security in DomainB for DomainC
6. Set cross-domain security in DomainC for DomainB

Because DomainD does not participate in the transaction, using cross-domain credential mapping is not required. However, see Adding Domains to the Exclude List Based on Transaction Participation for further clarification.

To present this information in another way, consider Table 7-2. A table cell containing **Yes** indicates that you must configure cross domain security for this domain combination.

Table 7-2 Setting Cross Domain Security with Three Participating Domains

—	DomainA	DomainB	DomainC	DomainD
DomainA	NA	**Yes**	**Yes**	No
DomainB	**Yes**	NA	**Yes**	No
DomainC	**Yes**	**Yes**	NA	No
DomainD	No	No	No	NA

If you were then to add DomainD and leave a DomainE out of the cross-domain security configuration, the cross-domain credential map would be as shown in Table 7-3. A table cell containing **Yes** indicates that you must configure cross domain security for this domain combination.

Table 7-3 Setting Cross Domain Security with Five Participating Domains

—	DomainA	DomainB	DomainC	DomainD	DomainE
DomainA	NA	**Yes**	**Yes**	**Yes**	No
DomainB	**Yes**	NA	**Yes**	**Yes**	No
DomainC	**Yes**	**Yes**	NA	**Yes**	No

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Setting Cross Domain Security with Five Participating Domains

—	DomainA	DomainB	DomainC	DomainD	DomainE
DomainD	**Yes**	**Yes**	**Yes**	NA	No
DomainE	No	No	No	No	NA

Adding Domains to the Exclude List Based on Transaction Participation

The exclude list provides a mechanism for a server in a domain with Cross Domain Security configured to participate in a transaction with a server in another domain that does not support or have Cross Domain Security enabled.

If any server in a domain in which cross domain security is not configured participates in a transaction with any server in a domain in which cross domain security is configured, add that domain to the exclude list of the domain that has cross domain security configured.

You do not need to add the domain to the exclude list of all domains that have cross domain security configured; the domain must explicitly participate in a transaction with the domain in question for this requirement to take effect.

> **Note**
>
> You may find it more convenient to add the names of all domains for which cross-domain security is not enabled to the list of excluded domains. If you exclude only those domains that do not participate in transactions, the exclusions may be sufficient only in the case of transactions. As described in ”Excluding Domains From Cross-Domain Security”, in Administering Security for Oracle WebLogic Server the more common use case is to exclude all domains for which cross-domain security is not enabled.

Consider the following scenario:

- Transaction #1:
  - DomainA has Server1 (coordinator)
  - DomainB has Server2 (sub-coordinator)
  - DomainC has Server3 and Server4 (Server3 is a sub-coordinator)
  - DomainD has Server5 (does not participate in the transaction, cross-domain security not configured)
- Transaction #2:
  - DomainB has Server6 (coordinator)
  - DomainD has Server5 (sub-coordinator, cross-domain security not configured)

In this case DomainD has to be in the exclusion list of DomainB because of Transaction #2.

You do not need to include it in the exclusion list of DomainA or DomainC because DomainD does not participate in any transactions with servers in these two domains.

Copyright © 2007, 2026, Oracle and/or its affiliates.

8

Java Transaction API and Oracle WebLogic Extensions

Learn about the Java Transaction API (JTA) and extensions to the API provided by Oracle.

- JTA API Overview
  Learn about the JTA API with the help of brief overview. WebLogic Server supports the `jakarta.transaction` package and the `javax.transaction.xa` package, which implement the Java Transaction API (JTA) for Java applications.
- Oracle WebLogic Extensions to JTA
  Extensions to the Java Transactions API are provided where the JTA specification does not cover implementation details and where additional capabilities are required.

JTA API Overview

Learn about the JTA API with the help of brief overview. WebLogic Server supports the `jakarta.transaction` package and the `javax.transaction.xa` package, which implement the Java Transaction API (JTA) for Java applications.

For more information about JTA, see the Java Transaction API (JTA) Specification published at `https://jakarta.ee/specifications/transactions/2.0/`. For a detailed description of the `jakarta.transaction` and `javax.transaction.xa` interfaces, see the JTA Javadoc.

JTA includes the following components:

- An interface for demarcating and controlling transactions from an application, `jakarta.transaction.UserTransaction`. You use this interface as part of a Java client program or within an EJB as part of a bean-managed transaction.
- An interface for allowing a transaction manager to demarcate and control transactions for an application, `jakarta.transaction.TransactionManager`. This interface is used by an EJB container as part of a container-managed transaction and uses the `jakarta.transaction.Transaction` interface to perform operations on a specific transaction.
- Interfaces that allow the transaction manager to provide status and synchronization information to an applications server, `jakarta.transaction.Status` and `jakarta.transaction.Synchronization`. Applications can use status constants and register synchronization callbacks through the transaction APIs, subject to component restrictions.
- Interfaces for allowing a transaction manager to work with resource managers for XA-compliant resources (`javax.transaction.xa.XAResource`) and to retrieve transaction identifiers (`javax.transaction.xa.Xid`). Resource-manager and gateway integrations use these interfaces to participate in transaction coordination, as described in Coordinating XAResources with the WebLogic Server Transaction Manager and Participating in Transactions Managed by a Third-Party Transaction Manager.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Oracle WebLogic Extensions to JTA

Extensions to the Java Transactions API are provided where the JTA specification does not cover implementation details and where additional capabilities are required.

Oracle WebLogic provides the following capabilities based on interpretations of the JTA specification:

- Client-initiated transactions—the JTA transaction manager interface (`jakarta.transaction.TransactionManager`) is made available to clients and bean providers through JNDI. This allows clients and EJBs using bean-managed transactions to suspend and resume transactions.

  > **Note**
  >
  > A suspended transaction must be resumed in the same server process in which it was suspended.

- Scope of transactions—transactions can operate within and between clusters and domains.
- Enhanced `jakarta.transaction.TransactionSynchronizationRegistry` support—WebLogic Server provides the ability to lookup the `TransactionSynchronizationRegistry` object in JNDI using the standard name of `java:comp/TransactionSynchronizationRegistry`. Oracle extends support by providing two additional global JNDI names: `jakarta/transaction/TransactionSynchronizationRegistry` and `weblogic/transaction/TransactionSynchronizationRegistry`. See `jakarta.transaction.TransactionSynchronizationRegistry` at https://jakarta.ee/specifications/transactions/2.0/apidocs/jakarta/transaction/transactionsynchronizationregistryl.

Oracle WebLogic Server provides the following classes and interfaces as extensions to JTA:

- `weblogic.transaction.RollbackException` (extends `jakarta.transaction.RollbackException`)

  This class preserves the original reason for a rollback for use in more comprehensive exception information.

- `weblogic.transaction.TransactionManager` (extends `jakarta.transaction.TransactionManager`)

  The WebLogic JTA transaction manager object supports this interface, which allows XA resources to register and unregister themselves with the transaction manager on startup. It also allows a transaction to be resumed after suspension.

  This interface includes the following methods:

  - `registerStaticResource`, `registerDynamicResource`, and `unregisterResource`
  - `registerResource`— (new in WebLogic Server 8.1) This method includes support for properties that determine how the resource is controlled by the transaction manager.
  - `getTransaction`
  - `forceResume` and `forceSuspend`
  - `begin`

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `weblogic.transaction.Transaction` (extends `jakarta.transaction.Transaction`)

  The WebLogic JTA transaction object supports this interface, which allows users to get and set transaction properties.

  This interface includes the following methods:

  - `setName` and `getName`
  - `addProperties`, `setProperty`, `getProperty`, and `getProperties`
  - `setRollbackReason` and `getRollbackReason`
  - `getHeuristicErrorMessage`
  - `getXID` and `getXid`
  - `getStatusAsString`
  - `getMillisSinceBegin`
  - `getTimeToLiveMillis`
  - `isTimedOut`

- `weblogic.transaction.TransactionHelper`

  This class enables you to obtain the current transaction manager and transaction. It replaces `TxHelper`.

  This interface includes the following static methods:

  - `getTransaction`
  - `getUserTransaction`
  - `getTransactionManager`

- `weblogic.transaction.TxHelper` (Deprecated, use `TransactionHelper` instead)

  This class enables you to obtain the current transaction manager and transaction.

  This interface includes the following static methods:

  - `getTransaction`, `getUserTransaction`, `getTransactionManager`
  - `status2String`

- `weblogic.transaction.XAResource` (extends `javax.transaction.xa.XAResource`)

  This class provides delistment capabilities for XA resources.

  This interface includes the following method:

  - `getDelistFlag`

- `weblogic.transaction.nonxa.NonXAResource`

  This interface enables resources that do not support the `javax.transaction.xa.XAResource` interface to easily integrate with the WebLogic Server transaction manager. The transaction manager supports a variation of the Last Agent two-phase commit optimization that allows a non-XA resource to participate in a distributed transaction. The protocol issues a two-phase commit to the non-XA resource and uses the result of the operation to base the commit decision for the transaction.

For a detailed description of the WebLogic extensions to the `jakarta.transaction` and `javax.transaction.xa` interfaces, see the `weblogic.transaction` package summary in the *Java API Reference for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

9

Logging Last Resource Transaction Optimization

Understand how WebLogic Server supports 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.

In a global two-phase commit (2PC) transaction with an LLR participant, the WebLogic Server transaction manager follows these basic steps:

- Calls prepare on all other (XA-compliant) transaction participants.
- Inserts a commit record to a table on the LLR participant (rather than to the file-based transaction log).
- Commits the LLR participant's local transaction (which includes both the transaction commit record insert and the application's SQL work).
- Calls commit on all other transaction participants.
- After the transaction completes successfully, lazily deletes the database transaction log entry as part of a future transaction.

This chapter includes the following sections:

For more information about the advantages of LLR, see Understanding the Logging Last Resource Transaction Option in *Administering JDBC Data Sources for Oracle WebLogic Server*.

- About the LLR Optimization Transaction Optimization
  When there is one database participant in a 2PC transaction, the Logging Last Resource (LLR) Optimization transaction option can significantly improve transaction performance by eliminating some XA overhead for database processing and by avoiding the use of JDBC XA drivers, which typically are less efficient than non-XA drivers.
- Logging Last Resource Processing Details
  At server boot or data source deployment, LLR data sources load or create a table on the database from which the data source pools database connections. The table is created in the schema determined by the user specified to create database connections. If the database table cannot be created or loaded, then server boot fails.
- LLR Database Table Details
- Failure and Recovery Processing for LLR
  Learn how WebLogic transaction manager processes transaction failures for LLR.
- Optimizing Performance with LLR
  Learn how to optimize performance with LLR by running applications directly on the coordinating server and use connection instances that are directly hosted on the

Copyright © 2007, 2026, Oracle and/or its affiliates.

coordinator. Also, to improve performance in environments using Oracle RAC, you can specify a LLR table for each data source instead of for each server to better utilize the local node caches in Oracle RAC clusters.

- First Resource Commit Ordering
  To specify a resource as the first resource committed in a transaction, set the connection pool driver boolean property `weblogic.jdbc.firstResourceCommit` to `true`.

About the LLR Optimization Transaction Optimization

When there is one database participant in a 2PC transaction, the Logging Last Resource (LLR) Optimization transaction option can significantly improve transaction performance by eliminating some XA overhead for database processing and by avoiding the use of JDBC XA drivers, which typically are less efficient than non-XA drivers.

In many cases a global transaction becomes a two-phase commit (2PC) transaction because it involves a database operation (using JDBC) and another non-database operation, such as a message queueing operation (using JMS). In cases such as this where there is one database participant in a 2PC transaction, the Logging Last Resource (LLR) Optimization transaction option can significantly improve transaction performance by eliminating some XA overhead for database processing and by avoiding the use of JDBC XA drivers, which typically are less efficient than non-XA drivers. The LLR transaction option does not incur the same data risks as borne by the Emulate Two-Phase Commit JDBC data source option and the NonXAResource resource adapter (Connector) option.

Logging Last Resource Processing Details

At server boot or data source deployment, LLR data sources load or create a table on the database from which the data source pools database connections. The table is created in the schema determined by the user specified to create database connections. If the database table cannot be created or loaded, then server boot fails.

Within a global transaction, the first connection obtained from an LLR data source reserves an internal JDBC connection that is dedicated to the transaction. The internal JDBC connection is reserved on the specific server that is also the transactions' coordinator. All subsequent transaction operations on any connections obtained from a same-named data source on any server are routed to this same single internal JDBC connection.

When an LLR transaction is committed, the WebLogic Server transaction manager handles the processing transparently. From an application perspective, the transaction semantics remain the same, but from an internal perspective, the transaction is handled differently than standard XA transactions. When the application commits the global transaction, the WebLogic Server transaction manager atomically commits the local transaction on the LLR connection before committing transaction work on any other transaction participants. For a two-phase commit transaction, the transaction manager also writes a 2PC record on the database as part of the same local transaction. After the local transaction completes successfully, the transaction manager calls commit on all other global transaction participants. After all other transaction participants complete the commit phase, the related LLR 2PC transaction record is freed for deletion. The transaction manager lazily deletes the transaction record after a short interval or with another local transaction.

If the application rolls back the global transaction or the transaction times out, the transaction manager rolls back the work in the local transaction and does not store a 2PC record in the database.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Also see Understanding the Logging Last Resource Transaction Option in *Administering JDBC Data Sources for Oracle WebLogic Server*.

For a list of data source configuration and usage requirements and limitations, see the following topics in *Administering JDBC Data Sources for Oracle WebLogic Server*:

- Programming Considerations and Limitations for LLR Data Sources
- Administrative Considerations and Limitations for LLR Data Sources

LLR Database Table Details

Each WebLogic server instance maintains a database "LLR" table on the database to which a JDBC LLR data source pools database connections. These tables are used for storing transaction log records, and are automatically created.If multiple LLR data sources are deployed on the same WebLogic server instance and connect to the same database instance and database schema, they also share the same LLR table.
LLR table names are automatically generated unless administrators choose to configure them. The default table name is `WL_LLR_SERVERNAME`. For some DBMS systems, the maximum length for a table name is 18 characters. You should consider maximum table name length when configuring your environment.

Note the following restrictions regarding LLR database tables:

- The server **does not boot** if an LLR table is unreachable during boot. LLR transaction records must be available to correctly resolve in-doubt transactions during recovery, which runs automatically at server startup.
- Multiple servers must not share the same LLR table. On server startup, WebLogic Server checks to ensure that the domain and server name of the JDBC data source match the domain and server name stored in the table when the table is created. If WebLogic Server detects that multiple servers are sharing the same LLR table, WebLogic Server instance shuts down one or more of the servers.

To change the table name used to store transaction log records for the resource, follow these steps:

1. On the Servers: Configuration: General page, click **Show Advanced Fields** to show the advanced configuration options.
2. In JDBC LLR Table Name, enter the name of the table to use to store transaction records for the resource, then click Save.

> **Note**
>
> The table names while using dynamic servers needs to follow certain considerations. See, Limitations and Considerations When Using Dynamic Clusters.

3. Repeat steps 2 and 3 for each server on which the LLR-enabled data source is deployed.
4. Click Activate Changes in the Change Center.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> You must restart all servers for the change to take effect.

- LLR Table Transaction Log Records

LLR Table Transaction Log Records

For each committed 2PC LLR transaction, the transaction manager automatically inserts a transaction record into an LLR database table. Once LLR transactions complete, the transaction manager lazily deletes their transaction records. If an LLR table transaction log record delete fails, the server logs a warning message and retry the delete again later.

If you move a database that contains LLR transaction records, ensure that you move the LLR table contents to the new database so that transactions can be completed properly.

> **Note**
>
> Do not manually delete the LLR transaction records or the LLR table in a production system. Doing so can lead to silent heuristic transaction failures which are not logged.

Failure and Recovery Processing for LLR

Learn how WebLogic transaction manager processes transaction failures for LLR.

In general, the WebLogic transaction manager processes transaction failures in the following way:

- For two-phase commit errors that occur before the local transaction commit is attempted, the transaction manager immediately throws a transaction rolled back exception.
- For two-phase commit errors that occur during the local transaction commit, the behavior depends on whether the transaction record is written to the database:
  - If the record is written, the transaction manager commits the transaction.
  - If the record is not written, the transaction manager rolls back the transaction.
  - If it is unknown whether the record is written, the transaction manager throws an ambiguous commit failure exception and attempts to complete the transaction every 5 seconds until the transaction abandon timeout. If the transaction is still incomplete, the transaction manager logs an abandoned transaction message.
- Coordinating Server Crash
- JDBC Connection Failure
- LLR Transaction Recover During Server Startup
- Failover Considerations for LLR

Coordinating Server Crash

If a transaction's coordinating server crashes before an LLR resource stores its transaction log record or before an LLR resource commits, the transaction rolls back. If the server crashes

Copyright © 2007, 2026, Oracle and/or its affiliates.

after the LLR resource is committed, the transactions eventually fully commit. During server boot, the transaction coordinator uses the LLR resource to read the transaction log record from the database and then use the recovered information to commit any unfinished work on any participating non-LLR XA resources.

JDBC Connection Failure

If the JDBC connection in an LLR resource fails during a 2PC transaction record insert, the transaction manager rolls back the transaction.

If the JDBC connection in an LLR resource fails during the commit of the local transaction, the result depends on whether the transaction is a one-phase commit (1PC, where the LLR resource is the only participant) or 2PC:

- For a 1PC transaction, the transaction are fully committed, fully rolled back, or block waiting for the resolution of the local transaction. The outcome of the transaction is fully ACID because it is eventually fully committed or fully rolled back.
- For a 2PC transaction, the outcome is as described in Failure and Recovery Processing for LLR.

LLR Transaction Recover During Server Startup

During server startup, the transaction manager for each WebLogic server must recover incomplete transactions coordinated by the server, including LLR transactions. To do so, each server attempts to read the transaction records from the LLR database tables for each LLR data source. If the server cannot access the LLR database tables or if the recovery fails, the server instance does not start and the transaction manager marks the server with a bad health state: HealthState.HEALTH_FAILED.

If a timeout occurs during recovery, it may be due to unresolved local transactions that have locked rows within the LLR log tables. Such local transactions must be resolved so that the transaction manager can determine the state of the global transaction whose record is stored in the locked row. Local database transactions can only be diagnosed and resolved using each database's specific tools (the commands differ from database to database).

Failover Considerations for LLR

Consider the following notes and limitations regarding failover with LLR:

- A transaction log (TLog) is still required for LLR transactions:
  - TLog still stores transaction manager "checkpoint" records
  - TLog must still be reachable or copied on failover
- 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.

Optimizing Performance with LLR

Learn how to optimize performance with LLR by running applications directly on the coordinating server and use connection instances that are directly hosted on the coordinator. Also, to improve performance in environments using Oracle RAC, you can specify a LLR table

Copyright © 2007, 2026, Oracle and/or its affiliates.

for each data source instead of for each server to better utilize the local node caches in Oracle RAC clusters.

This section includes the following information:

- Optimizing Transaction Coordinator Location
- Varied Performance for Read-Only Operations Through an LLR Data Source
- Dedicating LLR Tables by Data Source

Optimizing Transaction Coordinator Location

Within a global transaction with an LLR participant, WebLogic Server automatically routes all connection operations to the transaction's coordinating server. This routing can be expensive. You may see better performance if you optimize your applications to run directly on the coordinating server if possible, and optimize your applications to use connection instances that are directly hosted on the coordinator.

For client applications that begin a transaction, the coordinator of transaction is the first WebLogic server the client calls under the transaction (any RMI, EJB, JDBC, or JMS call). In the JMS case, this is the server that hosts the client's JMS connection, which is not necessarily the same as the server that hosts the JMS destination.

For server side applications, the coordinator of the transaction is the local server if a local resource is invoked first (including JMS destinations and JDBC connections) unless a remote server is called first (any remotely hosted JDBC connection, EJB, RMI call, or JMS connection). This includes remote servers in other clusters or domains.

Varied Performance for Read-Only Operations Through an LLR Data Source

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

Dedicating LLR Tables by Data Source

To improve performance in environments using Oracle RAC, you can specify a LLR table for each data source instead of for each server to better utilize the local node caches in Oracle RAC clusters.

Use the following system property to set the specification of a LLR table by data source when starting a WebLogic Server instance:

`-Dweblogic.llr.table.`*`datasourcename`*`=`*`tablename`*

where: *`datasourcename`* is the name of a data source and *`tablename`* is the name of the LLR table that maps to *`datasourcename`*.

For example, using the system property:

```
-Dweblogic.llr.table.LLRDS1=myllrtable1
```

When the server starts:

- a `INFO` message is written to `stdout`

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
LLR data source LLRDS1 using LLR table myllrtable1
```

- All LLR entries for the server that use data source LLRDS1 are stored in the LLR table named `mylltable1`.

Define one table for each data source on each server on which the data source is targeted. The same table cannot be shared by different WLS instances. If LLRDS1 is targeted to two WebLogic Server instances S1 and S2, then create two tables: S1_LLRDS1 and S2_LLRDS1, and specify the proper system properties for each server.

For example:

For instance S1, use `-Dweblogic.llr.table.LLRDS1=S1_LLRDS1`

For instance S2, use `-Dweblogic.llr.table.LLRDS1=S2_LLRDS1`

> **Note**
>
> In the next WebLogic Server release, the `node-id` will automatically be captured in an extended LLR table allowing data to be partitioned to the respective WebLogic Server node without needing to manually allocate a table per data source.

- Limitations

Limitations

JTA service migration does not support LLR tables that are dedicated by data source.

First Resource Commit Ordering

To specify a resource as the first resource committed in a transaction, set the connection pool driver boolean property `weblogic.jdbc.firstResourceCommit` to `true`.

A LLR resource cannot be used for resource commit ordering. In such a scenario, if you still want to use resource commit ordering, an XA datasource can be used instead of a the local transaction-based LLR non-XA resource.

Therefore, to specify a resource as the first resource committed in a transaction, set the connection pool driver boolean property `weblogic.jdbc.firstResourceCommit` to `true`.

When registering a data source that has `weblogic.jdbc.firstResourceCommit` set to `true`, the data source container will set the `weblogic.transaction.first.resource.commit` registration property to `true` when registering with the transaction manager.

See Configuring Services in *Oracle WebLogic Remote Console Online Help*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

10

Transactions in EJB Applications

Learn how to integrate transactions in Jakarta Enterprise Beans (EJBs) applications that run under Oracle WebLogic Server.

- Before You Begin
  Before proceeding with transactions in WebLogic Server EJB applications, you should be familiar with the concepts of the EJB transactions and its implementation with the help of sample code fragments provided for EJB applications.
- General Guidelines
  Understand the guidelines for implementing transactions in EJB applications for WebLogic Server.
- Transaction Attributes
  Learn about the transaction attributes for container-managed and bean-managed transactions.
- Participating in a Transaction
  Understand usage of the term "participating in a transaction", which is generally used when bean is invoked in container-managed transaction or bean-managed transaction.
- Transaction Semantics
  Learn about the transaction semantics for container-managed and bean-managed transactions.
- Session Synchronization
  A stateful session bean using container-managed transactions can implement the `jakarta.ejb.SessionSynchronization` interface to provide transaction synchronization notifications.
- Synchronization During Transactions
  If a bean implements `SessionSynchronization`, the WebLogic Server EJB container makes few callbacks to the bean during the transaction time. A synchronization cycle can occur when a registered object receives a `beforeCompletion` callback.
- Setting Transaction Timeouts
  Bean providers can specify the timeout period for transactions in EJB applications. If the duration of a transaction exceeds the specified timeout setting, then the Transaction Service rolls back the transaction automatically.
- Handling Exceptions in EJB Transactions
  WebLogic Server EJB applications must catch and handle specific exceptions thrown during transactions.

Before You Begin

Before proceeding with transactions in WebLogic Server EJB applications, you should be familiar with the concepts of the EJB transactions and its implementation with the help of sample code fragments provided for EJB applications.

Before you begin, you should read Introducing Transactions, particularly the following topics:

- Transactions in WebLogic Server EJB Applications

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Transactions Sample EJB Code

This chapter supplements Jakarta Enterprise Beans 4.0; it also includes compatibility material for the Enterprise Beans 2.x programming model.

> **Note**
>
> Before proceeding with the rest of this chapter, read Jakarta Enterprise Beans 4.0, Chapter 8, *Support for Transactions*.

For information about implementing Jakarta Enterprise Beans in WebLogic Server applications, see *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

General Guidelines

Understand the guidelines for implementing transactions in EJB applications for WebLogic Server.

The following general guidelines apply when implementing transactions in EJB applications for WebLogic Server:

- The EJB specification allows for flat transactions only. Transactions cannot be nested.
- The EJB specification allows for distributed transactions that span multiple resources (such as databases) and supports the two-phase commit protocol for both EJB CMP 2.1 and EJB CMP 1.1.
- Use standard programming techniques to optimize transaction processing. For example, properly demarcate transaction boundaries and complete transactions quickly.
- Use a database connection from a local TxDataSource—on the WebLogic Server instance on which the EJB is running. Do not use a connection from a TxDataSource on a remote WebLogic Server instance.
- Be sure to tune the EJB cache to ensure maximum performance in transactional EJB applications. See *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

For general guidelines about the WebLogic Server Transaction Service, see Capabilities and Limitations.

Transaction Attributes

Learn about the transaction attributes for container-managed and bean-managed transactions.

- About Transaction Attributes for EJBs
- Transaction Attributes for Container-Managed Transactions
- Transaction Attributes for Bean-Managed Transactions

About Transaction Attributes for EJBs

Transaction attributes determine how transactions are managed in EJB applications. For each EJB, the transaction attribute specifies whether transactions are demarcated by the WebLogic Server EJB container (container-managed transactions) or by the EJB itself (bean-managed transactions). The setting of the `transaction-type` element in the deployment descriptor determines whether an EJB is container-managed or bean-managed. See Chapter 8, "Support

Copyright © 2007, 2026, Oracle and/or its affiliates.

for Transactions," and Chapter 13, "Deployment Descriptor," in the EJB Specification 4.0, for more information about the `transaction-type` element.

In general, the use of container-managed transactions is preferred over bean-managed transactions because application coding is simpler. For example, in container-managed transactions, transactions do not need to be started explicitly.

WebLogic Server fully supports method-level transaction attributes as defined in Section 8.6.3 in the EJB Specification 4.0.

Transaction Attributes for Container-Managed Transactions

For container-managed transactions, the transaction attribute is specified in the `container-transaction` element in the deployment descriptor. Container-managed transactions include all entity beans and any stateful or stateless session beans with a `transaction-type` set to `Container`. For more information about these elements, see *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

The Application Assembler can specify the following transaction attributes for EJBs and their business methods:

- `NotSupported`
- `Supports`
- `Required`
- `RequiresNew`
- `Mandatory`
- `Never`

For a detailed explanation about how the WebLogic Server EJB container responds to the `trans-attribute` setting, see section 8.7.3.2 in the EJB Specification 4.0 at https://jakarta.ee/specifications/enterprise-beans/4.0/jakarta-enterprise-beans-spec-core-4.0.

The WebLogic Server EJB container automatically sets the transaction timeout if a timeout value is not defined in the deployment descriptor. The container uses the value of the `Timeout Seconds` configuration parameter. The default timeout value is 30 seconds.

For EJBs with container-managed transactions, the EJBs have no access to the `jakarta.transaction.UserTransaction` interface, and the entering and exiting transaction contexts must match. In addition, EJBs with container-managed transactions have limited support for the `setRollbackOnly` and `getRollbackOnly` methods of the `jakarta.ejb.EJBContext` interface, where invocations are restricted by rules specified in Sections 8.6.3.8 and 8.6.3.9 of the EJB Specification 4.0.

Transaction Attributes for Bean-Managed Transactions

For bean-managed transactions, the bean specifies transaction demarcations using methods in the `jakarta.transaction.UserTransaction` interface. Bean-managed transactions include any stateful or stateless session beans with a `transaction-type` set to `Bean`. Entity beans cannot use bean-managed transactions.

For stateless session beans, the entering and exiting transaction contexts must match. For stateful session beans, the entering and exiting transaction contexts may or may not match. If they do not match, the WebLogic Server EJB container maintains associations between the bean and the non-terminated transaction.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Session beans with bean-managed transactions cannot use the `setRollbackOnly` and `getRollbackOnly` methods of the `jakarta.ejb.EJBContext` interface.

Participating in a Transaction

Understand usage of the term "participating in a transaction", which is generally used when bean is invoked in container-managed transaction or bean-managed transaction.

When the EJB Specification 2.1 uses the phrase "participating in a transaction", Oracle interprets this to mean that the bean meets either of the following conditions:

- The bean is invoked in a transactional context (container-managed transaction).
- The bean begins a transaction using the UserTransaction API in a bean method invoked by the client (bean-managed transaction), and it does *not* suspend or terminate that transaction upon completion of the corresponding bean method invoked by the client.

Transaction Semantics

Learn about the transaction semantics for container-managed and bean-managed transactions.

- Transaction Semantics for Container-Managed Transactions
- Transaction Semantics for Bean-Managed Transactions

The EJB Specification 2.1 describes semantics that govern transaction processing behavior based on the EJB type (entity bean, stateless session bean, or stateful session bean) and the transaction type (container-managed or bean-managed). These semantics describe the transaction context at the time a method is invoked and define whether the EJB can access methods in the `jakarta.transaction.UserTransaction` interface. EJB applications must be designed with these semantics in mind.

- Transaction Semantics for Container-Managed Transactions
- Transaction Semantics for Bean-Managed Transactions

Transaction Semantics for Container-Managed Transactions

For container-managed transactions, transaction semantics vary for each bean type.

- Transaction Semantics for Stateful Session Beans
- Transaction Semantics for Stateless Session Beans
- Transaction Semantics for Entity Beans

Transaction Semantics for Stateful Session Beans

Table 10-1 describes the transaction semantics for stateful session beans in container-managed transactions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 10-1 Transaction Semantics for Stateful Session Beans in Container-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
`Constructor`	Unspecified	No
`setSessionContext()`	Unspecified	No
`ejbCreate()`	Unspecified	No
`ejbRemove()`	Unspecified	No
`ejbActivate()`	Unspecified	No
`ejbPassivate()`	Unspecified	No
Business method	Yes or No based on transaction attribute	No
`afterBegin()`	Yes	No
`beforeCompletion()`	Yes	No
`afterCompletion()`	No	No

Transaction Semantics for Stateless Session Beans

Table 10-2 describes the transaction semantics for stateless session beans in container-managed transactions.

Table 10-2 Transaction Semantics for Stateless Session Beans in Container-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
`Constructor`	Unspecified	No
`setSessionContext()`	Unspecified	No
`ejbCreate()`	Unspecified	No
`ejbRemove()`	Unspecified	No
Business method	Yes or No based on transaction attribute	No

Transaction Semantics for Entity Beans

Table 10-3 describes the transaction semantics for entity beans in container-managed transactions.

Table 10-3 Transaction Semantics for Entity Beans in Container-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
Constructor	Unspecified	No
`setEntityContext()`	Unspecified	No
`unsetEntityContext()`	Unspecified	No

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 10-3 (Cont.) Transaction Semantics for Entity Beans in Container-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
`ejbCreate()`	Determined by transaction attribute of matching create	No
`ejbPostCreate()`	Determined by transaction attribute of matching create	No
`ejbRemove()`	Determined by transaction attribute of matching remove	No
`ejbFind()`	Determined by transaction attribute of matching find	No
`ejbActivate()`	Unspecified	No
`ejbPassivate()`	Unspecified	No
`ejbLoad()`	Determined by transaction attribute of business method that invoked `ejbLoad()`	No
`ejbStore()`	Determined by transaction attribute of business method that invoked `ejbStore()`	No
Business method	Yes or No based on transaction attribute	No

Transaction Semantics for Bean-Managed Transactions

For bean-managed transactions, the transaction semantics differ between stateful and stateless session beans. For entity beans, transactions are never bean-managed.

- Transaction Semantics for Stateful Session Beans
- Transaction Semantics for Stateless Session Beans

Transaction Semantics for Stateful Session Beans

Table 10-4 describes the transaction semantics for stateful session beans in bean-managed transactions.

Table 10-4 Transaction Semantics for Stateful Session Beans in Bean-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
Constructor	Unspecified	No
`setSessionContext()`	Unspecified	No
`ejbCreate()`	Unspecified	Yes
`ejbRemove()`	Unspecified	Yes
`ejbActivate()`	Unspecified	Yes
`ejbPassivate()`	Unspecified	Yes

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 10-4 (Cont.) Transaction Semantics for Stateful Session Beans in Bean-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
`Business method`	Typically, no unless a previous method execution on the bean had completed while in a transaction context	Yes
`afterBegin()`	Not applicable	Not applicable
`beforeCompletion()`	Not applicable	Not applicable
`afterCompletion()`	Not applicable	Not applicable

Transaction Semantics for Stateless Session Beans

Table 10-5 describes the transaction semantics for stateless session beans in bean-managed transactions.

Table 10-5 Transaction Semantics for Stateless Session Beans in Bean-Managed Transactions

Method	Transaction Context at the Time the Method Was Invoked	Can Access UserTransaction Methods?
Constructor	Unspecified	No
`setSessionContext()`	Unspecified	No
`ejbCreate()`	Unspecified	Yes
`ejbRemove()`	Unspecified	Yes
Business method	No	Yes

Session Synchronization

A stateful session bean using container-managed transactions can implement the `jakarta.ejb.SessionSynchronization` interface to provide transaction synchronization notifications.

In addition, all methods on the stateful session bean must support one of the following transaction attributes: `REQUIRES_NEW`, `MANDATORY` or `REQUIRED`. For more information about the `jakarta.ejb.SessionSynchronization` interface, see Section 6.5.3 in the EJB Specification 2.1.

Synchronization During Transactions

If a bean implements `SessionSynchronization`, the WebLogic Server EJB container makes few callbacks to the bean during the transaction time. A synchronization cycle can occur when a registered object receives a `beforeCompletion` callback.

The WebLogic Server EJB container typically makes the following callbacks to the bean during transaction commit time:

- `afterBegin()`

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `beforeCompletion()`
- `afterCompletion()`

The EJB container can call other beans or involve additional XA resources in the `beforeCompletion` method. The number of calls is limited by the `beforeCompletionIterationLimit` attribute. This attribute specifies how many cycles of callbacks are processed before the transaction is rolled back. A synchronization cycle can occur when a registered object receives a `beforeCompletion` callback and then enlists additional resources or causes a previously synchronized object to be reregistered. The iteration limit ensures that synchronization cycles do not run indefinitely.

Setting Transaction Timeouts

Bean providers can specify the timeout period for transactions in EJB applications. If the duration of a transaction exceeds the specified timeout setting, then the Transaction Service rolls back the transaction automatically.

> **Note**
>
> You must set the timeout before you `begin()` the transaction. Setting a timeout does not affect transaction transactions that have begun.

Timeouts are specified according to the transaction type:

- **Container-managed transactions**. The Bean Provider configures the trans-timeout-seconds attribute in the weblogic-ejb-jar.xml deployment descriptor.
- **Bean-managed transactions**. An application calls the `UserTransaction.setTransactionTimeout` method.

> **Note**
>
> DBMS provides a `DISTRIBUTED_LOCK_TIMEOUT` setting, specifying the duration for which the XA transactions can hold the locks. If this is lower than the JTA timeout, the DBMS may abort the transactions.

Handling Exceptions in EJB Transactions

WebLogic Server EJB applications must catch and handle specific exceptions thrown during transactions.

For detailed information about handling exceptions, see Chapter 9, "Exception Handling," in the EJB Specification 4.0.

For more information about how exceptions are thrown by business methods in EJB transactions, see the following tables in Section 9.3.1: Table 7 (for container-managed transactions) and Table 8 (for bean-managed transactions).

For a client's view of exceptions, see Section 9.4, particularly Section 9.4.1 (application exceptions), Section 9.4.2(`java.rmi.RemoteException`), Section 9.4.2.1

Copyright © 2007, 2026, Oracle and/or its affiliates.

(`jakarta.transaction.TransactionRolledbackException`), and Section 9.4.2.2 (`jakarta.transaction.TransactionRequiredException`).

Copyright © 2007, 2026, Oracle and/or its affiliates.

11

Transactions in RMI Applications

Learn about the guidelines and additional references for using transactions in RMI applications that run under Oracle WebLogic Server.
This chapter includes the following sections:

- Before You Begin
  Before proceeding with transactions in WebLogic Server RMI applications, you should be familiar with the concepts of the RMI transactions and its implementation.
- General Guidelines
  There are certain guidelines for implementing transactions in RMI applications for WebLogic Server.

Before You Begin

Before proceeding with transactions in WebLogic Server RMI applications, you should be familiar with the concepts of the RMI transactions and its implementation.

Before you begin, read Introducing Transactions, particularly the following topics:

- Transactions in WebLogic Server RMI Applications
- Transactions Sample RMI Code

For more information about RMI applications, see *Developing Stand-alone Clients for Oracle WebLogic Server*.

General Guidelines

There are certain guidelines for implementing transactions in RMI applications for WebLogic Server.

- WebLogic Server allows for flat transactions only. Transactions cannot be nested.
- Use standard programming techniques to optimize transaction processing. For example, properly demarcate transaction boundaries and complete transactions quickly.
- For RMI applications, callback objects are not recommended for use in transactions because they are not subject to WebLogic Server administration.

  By default, all method invocations on the remote objects are transactional. If a callback object is required, you must compile these classes using the WebLogic RMI compiler using the -nontransactional flag. See Using the WebLogic RMI Compiler and WebLogic RMI Compiler Options in *Developing RMI Applications for Oracle WebLogic Server*.
- In RMI applications, an RMI client can initiate a transaction, but all transaction processing must occur on server objects or remote objects hosted by WebLogic Server. Remote objects hosted on a client JVM cannot participate in the transaction processing.

  As a work-around, you can suspend the transaction before making a call to a remote object on a client JVM, and then resume the transaction after the remote operation returns.

For general guidelines about the WebLogic Server Transaction Service, see Capabilities and Limitations.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Copyright © 2007, 2026, Oracle and/or its affiliates.

12

Using JDBC XA Drivers with WebLogic Server

Get an overview of XA JDBC drivers with WebLogic Server in distributed transactions. These drivers provide connectivity between WebLogic Server connection pools and the DBMS. Drivers used in distributed transactions are designated by the driver name followed by /XA; for example, Oracle Thin/XA Driver.
An XA data source DB user requires certain DBA table grants in order to recover pending transaction branches from the Oracle database. We now provide the ability to use two different DB credentials one for XA recover processing (with DBA privileges) and different DB credentials for application processing (no DBA privileges).

Data Source configuration allows specification of a database user and password that are used to create JDBC connections for XA recover processing. These credentials can be different from the database user and password used for JDBC application data. Configuration of separate XA recover credentials is optional and existing data source configurations will continue to support XA recovery as long as the database user is granted the appropriate permissions.

The XA recover credentials will be specified in a XA data source configuration using the WLS-internal driver properties weblogic.jdbc.xaRecoverUser and weblogic.jdbc.xaRecoverPassword. If XA recover credential properties are defined in a non-XA data source configuration they are ignored.

- Using Oracle Thin/XA Driver
  WebLogic Server ships with the Oracle Thin Driver pre-configured and ready to use.
- Using Other XA Drivers
  To use other XA-compliant JDBC drivers, you must include the path to the driver class libraries in your `CLASSPATH` and follow the configuration instructions provided by the vendor.

Using Oracle Thin/XA Driver

WebLogic Server ships with the Oracle Thin Driver pre-configured and ready to use.

If you want to update the driver or use a different version, see Using API Extensions in JDBC Drivers in *Developing JDBC Applications for Oracle WebLogic Server*.

- Set the Environment for the Oracle Thin/XA Driver
- Oracle Thin/XA Driver Configuration Properties

Set the Environment for the Oracle Thin/XA Driver

The following sections explain how to set the environment for the Oracle Thin/XA Driver.

- Configure WebLogic Server
- Enable XA on the Database Server

Configure WebLogic Server

For information on how to configure WebLogic Server for use with the Oracle Thin/XA Driver, see Using API Extensions in JDBC Drivers in *Developing JDBC Applications for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Enable XA on the Database Server

To prepare the database for XA, perform these steps:

1. Log on to sqlplus as system user, for example, `sqlplus sys/password@<DATABASE ALIAS NAME>`
2. Execute the following command: `@xaview.sql`

   The `xaview.sql` script resides in the `$ORACLE_HOME/rdbms/admin` directory
3. All database users for which XA data sources are defined need to be granted the following permissions individually or through one or more roles:
   - `grant select on dba_pending_transactions to <user/role>;`
   - `grant execute on dbms_xa to <user/role>;`

   Where `<user/role>` is either a data base user or a role.

If the above steps are not performed on the database server, normal XA database queries and updates may work fine. However, when the WebLogic Server Transaction Manager performs recovery on a re-boot after a crash, recover for the Oracle resource fails with `XAER_RMERR`. Crash recovery is a standard operation for an XA resource.

Oracle Thin/XA Driver Configuration Properties

For information on how to configure data source properties, see Configuring JDBC Data Sources in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Using Other XA Drivers

To use other XA-compliant JDBC drivers, you must include the path to the driver class libraries in your `CLASSPATH` and follow the configuration instructions provided by the vendor.

- Using WebLogic-branded Data Direct Drivers
- Additional Considerations

Using WebLogic-branded Data Direct Drivers

For information on how to configure WebLogic Server for use with WebLogic-branded Data Direct drivers, see Using WebLogic-branded DataDirect Drivers in *Developing JDBC Applications for Oracle WebLogic Server*

Additional Considerations

Your data base vendor may provide instructions to perform steps to prepare your environment for XA. These steps may include how to:

- Enable your data base server for XA. This may include granting additional user permissions.
- Compensate for vendor-specific transactional behavior, such as threading, timeout, and rollback behavior.
- Use vendor-specific connection properties.

Consult your data base vendor documentation for specific details.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Copyright © 2007, 2026, Oracle and/or its affiliates.

13

Coordinating XAResources with the WebLogic Server Transaction Manager

Understand how external, third-party systems can participate in distributed transactions coordinated by the WebLogic Server transaction manager by registering a `javax.transaction.xa.XAResource` implementation with the WebLogic Server transaction manager. The WebLogic Server transaction manager then drives the XAResource as part of its Two-Phase Commit (2PC) protocol. This is referred to as "exporting transactions."
By exporting transactions, you can integrate third-party transaction managers with the WebLogic Server transaction manager if the third-party transaction manager implements the `XAResource` interface. With an exported transaction, the third-party transaction manager would act as a subordinate transaction manager to the WebLogic Server transaction manager.

WebLogic Server can also participate in distributed transactions coordinated by third-party systems (sometimes referred to as foreign transaction managers). The WebLogic Server processing occurs as part of the work of the external transaction. The third-party transaction manager then drives the WebLogic Server transaction manager as part of its commit processing. This is referred to as "importing transactions."

Details about coordinating third-party systems within a transaction (exporting transactions) are described in this section. Details about participating in transactions coordinated by third-party systems (importing transactions) are described in Participating in Transactions Managed by a Third-Party Transaction Manager. Note that WebLogic Server IIOP, WebLogic Tuxedo Connector (WTC) gateway, and Oracle Java Adapter for Mainframe (JAM) gateway internally use the same mechanism described in these chapters to import and export transactions in WebLogic Server.

This chapter includes the following sections:

- Overview of Coordinating Distributed Transactions with Foreign XAResources
  In order to participate in distributed transactions coordinated by the WebLogic Server transaction manager, third-party systems must implement the `javax.transaction.xa.XAResource` interface and then register its `XAResource` object with the WebLogic Server transaction manager.
- Registering an XAResource to Participate in Transactions
  In order to participate in distributed transactions coordinated by the WebLogic Server transaction manager, third-party systems must implement the `javax.transaction.xa.XAResource` interface and then register its `XAResource` object with the WebLogic Server transaction manager.
- Enlisting and Delisting an XAResource in a Transaction
  For an XAResource to participate in a distributed transaction, the XAResource instance must be enlisted with the Transaction object. Depending on the enlistment mode, you may need to perform different actions.
- Commit processing
  During commit processing, the WebLogic Server transaction manager either uses the XAResource instances currently enlisted with the transaction, or the XAResource instances that are registered with the transaction manager to perform the two-phase commit.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Recovery
  When a WebLogic Server server is restarted, the WebLogic Server transaction manager reads its own transaction logs (with log records of transactions that are successfully prepared, but may not have completed the second commit phase of 2PC processing). The WebLogic Server transaction manager then continues to retry commit of the XAResources for these transactions.
- Resource Health Monitoring
  To prevent losing server threads to faulty XAResources, WebLogic Server JTA has an internal resource health monitoring mechanism. A resource is considered active if either there are no pending requests or the result from any of the XAResource pending requests is not `XAER_RMFAIL`. If an XAResource is not active within two minutes, the WebLogic Server transaction manager declares it dead.
- Jakarta EE Connector Architecture Resource Adapter
  Besides registering with the WebLogic Server transaction manager directly, you can also implement the Jakarta EE Connector Architecture resource adapter interfaces. When you deploy the resource adapter, the WebLogic Server Jakarta EE container registers the resource manager's XAResource with the WebLogic Server transaction manager automatically.
- Implementation Tips
  Learn some tips for exporting and importing transactions with the WebLogic Server transaction manager.
- Changes in the Resource Registration Name
  This release changes the behavior of the resource registration name for XA data source configurations. In previous releases, the JTA registration name was simply the name of the data source. Now, the registration name is a combination of data source name and domain.
- FAQs
  Get to know about the most frequently asked questions related to transactions.
- Additional Documentation about JTA
  Additional documentation for a connection-based Resource Usage scenario, which illustrates the JTA interaction between the transaction manager and resource manager is provided in the JTA specification.

Overview of Coordinating Distributed Transactions with Foreign XAResources

In order to participate in distributed transactions coordinated by the WebLogic Server transaction manager, third-party systems must implement the `javax.transaction.xa.XAResource` interface and then register its `XAResource` object with the WebLogic Server transaction manager.

For details about implementing the `javax.transaction.xa.XAResource` interface, refer to the Java Platform Enterprise Edition API Specifications at:

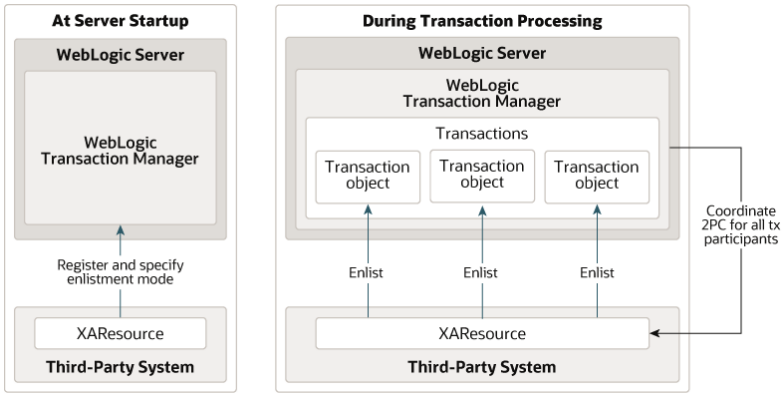
`https://docs.oracle.com/en/java/javase/17/docs/api/java.transaction.xa/javax/transaction/xa/XAResource.html`

During transaction processing, you must enlist the `XAResource` object of the third-party system with each applicable transaction object.

Figure 13-1 shows the process for third-party systems to participate in transactions coordinated by the WebLogic Server transaction manager.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Figure 13-1 Distributed Transactions with Third-Party Participants



Depending on the enlistment mode that you use when you enlist an XAResource object with a transaction, WebLogic Server may automatically delist the XAResource object at the appropriate time. For more information about enlistment and delistment, see Enlisting and Delisting an XAResource in a Transaction. For more information about registering XAResource objects with the WebLogic Server transaction manager, see Registering an XAReosurce to Participate in Transactions.

Registering an XAResource to Participate in Transactions

In order to participate in distributed transactions coordinated by the WebLogic Server transaction manager, third-party systems must implement the `javax.transaction.xa.XAResource` interface and then register its `XAResource` object with the WebLogic Server transaction manager.

Registration is required to:

- Specify the transaction branch qualifier for the `XAResource`. The branch qualifier identifies the transaction branch of the resource manager instance and is used for all distributed transactions that the resource manager (RM) instance participates in. Each transaction branch represents a unit of work in the distributed transaction and is isolated from other branches. Each transaction branch receives exactly one set of prepare-commit calls during Two-Phase Commit (2PC) processing. The WebLogic Server transaction manager uses the resource name as the transaction branch qualifier.

  A resource manager instance is defined by the `XAResource.isSameRM` method. XAResource instances that belong to the same resource manager instance should return true for `isSameRM`. Note that you should avoid registering the same resource manager instance under different resource names (for example, different resource branches) to avoid confusion of transaction branches.
- Specify the enlistment mode. For a resource manager instance to participate in a specific distributed transaction, it enlists an XAResource instance with the JTA `jakarta.transaction.Transaction` object. The WebLogic Server transaction manager provides three enlistment modes: static, dynamic, and object-oriented. Enlistment modes are discussed in greater detail in Enlisting and Delisting an XAResource in a Transaction.
- Bootstrap the `XAResource` if the WebLogic Server transaction manager must perform crash recovery. (The JTA Specification does not define a standard API to do so; see the Java Transaction API at `https://jakarta.ee/specifications/transactions/2.0/`).

Copyright © 2007, 2026, Oracle and/or its affiliates.

The Java Transaction API suggests that the transaction manager is responsible for assigning the branch qualifiers. However, for recovery to work properly, the same transaction branch qualifier must be supplied both at normal processing and upon crash recovery. As the transaction branch qualifier is specified during registration, registration with the WebLogic Server transaction manager is required to support crash recovery and normal transaction processing.

During recovery, the WebLogic Server transaction manager performs the following tasks:

- It reads its transaction log records and for those XA resources that participated in the distributed transactions that were logged, it continues the second phase of the 2PC protocol to commit the XA resources with the specified branch qualifier.
- It resolves any other in-doubt transactions of the XA resources by calling `XAResource.recover`. It then commits or rolls back the returned transactions (Xids) that belonged to it. (Note that the returned Xids would have the specified branch qualifier.)

> **Note**
>
> Registration is a per-process action (compared with enlistment and delistment which is per-transaction).

Failure to register the XAResource implementation with the WebLogic Server transaction manager may result in unexpected transaction branching behavior. If registration is not performed before the XA resource is enlisted with a WebLogic Server distributed transaction, the WebLogic Server transaction manager uses the class name of the XAResource instance as the resource name (and thus the branch qualifier), which may cause undesirable resource name and transaction branch conflicts.

Each resource manager instance should register itself only once with the WebLogic Server transaction manager. Each resource manager instance, as identified by the resource name during registration, adds significant overhead to the system during recovery and commit processing and health monitoring, increases memory used by associated internal data structures, reduces efficiency in searching through internal maps, and so forth. Therefore, for scalability and performance reasons, you should not indiscriminately register XAResource instances under different transaction branches.

Note that the JTA XAResource adopts an explicit transaction model, where the Xid is always explicitly passed in the XAResource methods and a single resource manager instance handles all of the transactions. This is in contrast to the CORBA OTS Resource, which adopts an implicit transaction model, where there is a different OTS Resource instance for each transaction that it participates in. You should use the JTA model when designing an XAResource.

Each foreign resource manager instance should register an XAResource instance with the WebLogic Server transaction manager upon server startup. In WebLogic Server, you can use startup classes to register foreign transaction managers.

Follow these steps to register the resource manager with the WebLogic Server transaction manager:

1. Obtain the WebLogic Server transaction manager using JNDI or the `TxHelper` interface:

```
import javax.transaction.xa.XAResource;
import weblogic.transaction.TransactionManager;
import weblogic.transaction.TxHelper;
InitialContext initCtx = ... ; // initialized to the initial context
TransactionManager tm = TxHelper.getTransactionManager();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

or

```
TransactionManager tm =
(TransactionManager)initCtx.lookup("weblogic.transaction.TransactionManager");
```

or

```
TransactionManager tm =
(TransactionManager)initCtx.lookup("jakarta.transaction.TransactionManager");
```

2. Register the XA resource instance with the WebLogic Server transaction manager:

```
String name = ... ; // name of the RM instance
XAResource res = ... ; // an XAResource instance of the RM instance
tm.registerResource(name, res); // register a resource with the standard enlistment
mode
```

or

```
tm.registerDynamicResource(name, res); // register a resource with the dynamic
enlistment mode
```

or

```
tm.registerStaticResource(name, res); // register a resource with the static
enlistment mode
```

Refer to Enlisting and Delisting an XAResource in a Transaction for a detailed discussion of the different enlistment modes. Note that when you register the XAResource, you specify the enlistment mode that is subsequently used, but you are not actually enlisting the resource during the registration process. Actual enlistment should be done with the transaction (not at server startup) using a different API, which is also discussed in detail in Enlisting and Delisting an XAResource in a Transaction.

Each XAResource instance that you register is used for recovery and commit processing of multiple transactions in parallel. Ensure that the XAResource instance supports resource sharing as defined in JTA Specification Version 1.0.1B Section 3.4.6.

> **Note**
>
> Duplicate registration of the same XAResource is ignored.

You should unregister the XAResource from the WebLogic Server transaction manager when the resource no longer accept new requests. Use the following method to unregister the XAResource:

```
tm.unregisterResource(name, res);
```

Enlisting and Delisting an XAResource in a Transaction

For an XAResource to participate in a distributed transaction, the XAResource instance must be enlisted with the Transaction object. Depending on the enlistment mode, you may need to perform different actions.

The WebLogic Server transaction manager supports the following enlistment modes:

- Standard Enlistment
- Dynamic Enlistment

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Static Enlistment

Even though you enlist the XAResource with the Transaction object, the enlistment *mode* is determined when you register the XAResource with the WebLogic Server transaction manger, not when you enlist the resource in the Transaction. See Registering an XAReosurce to Participate in Transactions.

`XAResource.start` and `end` calls can be expensive. The WebLogic Server transaction manager provides the following optimizations to minimize the number of these calls:

- Delayed delistment:

  Whether or not your XAResource implementation performs any explicit delistment or not, the WebLogic Server transaction manager always delays delisting of any XAResource instances that are enlisted in the current transaction until immediately before the following events, at which time the XAResource is delisted:

  – Returning the call to the caller, whether it is returned normally or with an exception

  – Making a call to another server

- Ignored duplicate enlistment:

  The WebLogic Server transaction manager ignores any explicit enlistment of an XAResource that is enlisted. This may happen if the XAResource is explicitly delisted (which is delayed or ignored by the WebLogic Server transaction manager as mentioned above) and is subsequently re-enlisted within the duration of the same call.

By default, the WebLogic Server transaction manager delists the XAResource by calling `XAResource.end` with the `TMSUSPEND` flag. Some database management systems may keep cursors open if `XAResource.end` is called with `TMSUSPEND`, so you may prefer to delist an XAResource by calling `XAResource.end` with `TMSUCCESS` wherever possible. To do so, you can implement the `weblogic.transaction.XAResource` interface (instead of the `javax.transaction.xa.XAResource`), which includes the `getDelistFlag` method. For more information, see `weblogic.transaction.XAResource` in the *Java API Reference for Oracle WebLogic Server*.

- Standard Enlistment
- Dynamic Enlistment
- Static Enlistment

Standard Enlistment

With standard enlistment mode, enlist the XAResource instance only once with the Transaction object. Also, it is possible to enlist multiple XAResource instances of the same branch with the same transaction. The WebLogic Server transaction manager ensures that `XAResource.end` is called on all XAResource instances when appropriate (as discussed below). The WebLogic Server transaction manager ensures that each branch receives only one set of prepare-commit calls during transaction commit time. However, attempting to enlist a particular XAResource instance when it is already enlisted is ignored.

Standard enlistment simplifies enlistment, but it may also cause unnecessary enlistment and delistment of an XAResource if the resource is not accessed at all within the duration of a particular method call.

To enlist an XAResource with the Transaction object, follow these steps:

1. Obtain the current Transaction object using the `TransactionHelper` interface:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
import weblogic.transaction.Transaction; // extends javax.transaction.Transaction
import weblogic.transaction.TransactionHelper;
Transaction tx = TransactionHelper.getTransaction();
```

2. Enlist the XAResource instance with the Transaction object:

```
tx.enlistResource(res);
```

After the XAResource is enlisted with the Transaction, the WebLogic Server transaction manager manages any subsequent delistment (as described in Enlisting and Delisting an XAResource in a Transaction) and re-enlistment. For standard enlistment mode, the WebLogic Server transaction manager re-enlists the XAResource in the same Transaction upon the following occasions:

- Before a request is executed
- After a reply is received from another server. (The WebLogic Server transaction manager delists the XAResource before sending the request to another server.)

Dynamic Enlistment

With the dynamic enlistment mode, you must enlist the XAResource instance with the Transaction object before every access of the resource. With this enlistment mode, only one XAResource instance from each transaction branch is allowed to be enlisted for each transaction at a time. The WebLogic Server transaction manager ignores attempts to enlist additional XAResource instances (of the same transaction branch) after the first instance is enlisted, but before it is delisted.

With dynamic enlistment, enlistments and delistments of XAResource instances are minimized.

The steps for enlisting the XAResource are the same as described in Standard Enlistment.

Static Enlistment

With static enlistment mode, you do not need to enlist the XAResource instance with any Transaction object. The WebLogic Server transaction manager implicitly enlists the XAResource for *all* transactions with the following events:

- Before a request is executed
- After a reply is received from another server

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> Consider the following before using the static enlistment mode:
>
> – Static enlistment mode eliminates the requirement to enlist XAResources. However, unnecessary enlistment and delistment may result, if the resource is not used in a particular transaction.
>
> – A faulty XAResource may adversely affect all transactions even if the resource is not used in the transaction.
>
> – A single XAResource instance is used to associate different transactions with different threads at the same time. That is, `XAResource.start` and `XAResource.end` can be called on the same XAResource instance in an interleaved manner for different Xids in different threads. You must ensure that the XAResource supports such an association pattern, which is not required by the JTA specification.
>
> Due to the performance overhead, poor fault isolation, and demanding transaction association requirement, static enlistment should only be used with discretion and after careful consideration.

Commit processing

During commit processing, the WebLogic Server transaction manager either uses the XAResource instances currently enlisted with the transaction, or the XAResource instances that are registered with the transaction manager to perform the two-phase commit.

The WebLogic Server transaction manager ensures that each transaction branch receives only one set of prepare-commit calls. You must ensure that any XAResource instance can be used for commit processing for multiple transactions simultaneously from different threads, as defined in JTA Specification Version 1.0.1B Section 3.4.6.

Recovery

When a WebLogic Server server is restarted, the WebLogic Server transaction manager reads its own transaction logs (with log records of transactions that are successfully prepared, but may not have completed the second commit phase of 2PC processing). The WebLogic Server transaction manager then continues to retry commit of the XAResources for these transactions.

As discussed in Registering an XAResource to Participate in Transactions, one purpose of the WebLogic Server transaction manager resource registration API is for bootstrapping XAResource instances for recovery. You must ensure that an XAResource instance is registered with the WebLogic Server transaction manager upon server restart. The WebLogic Server transaction manager retries the commit call every minute, until a valid XAResource instance is registered with the WebLogic Server transaction manager.

When a transaction manager that is acting as a transaction coordinator crashes, it is possible that the coordinator may not have logged some in-doubt transactions in the coordinator's transaction log. Thus, upon server restart, the coordinator must call `XAResource.recover` on the resource managers, and roll back the in-doubt transactions that were not logged. As with commit retries, the WebLogic Server transaction manager retries `XAResource.recover` every 5 minutes, until a valid XAResource instance is registered with the WebLogic Server transaction manager.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The WebLogic Server transaction manager checkpoints a new XAResource in its transaction log records when the XAResource is first enlisted with the WebLogic Server transaction manager. Upon server restart, the WebLogic Server transaction manager then calls `XAResource.recover` on all the resources previously checkpointed (removed from the transaction log records after the transaction completed). A resource is only removed from a checkpoint record if it has not been accessed for the last `PurgeResourceFromCheckpointIntervalSeconds` interval (default is 24 hours). Therefore, to reduce the resource recovery overhead, you should ensure that only a small number of resource manager instances are registered with the WebLogic Server transaction manager.

When implementing `XAResource.recover`, you should use the flags as described in the X/Open XA specification as follows:

- When the WebLogic Server transaction manager calls `XAResource.recover` with `TMSTARTRSCAN`, the resource returns the first batch of in-doubt Xids.

  The WebLogic Server transaction manager then calls `XAResource.recover` with `TMNOFLAGS` repeatedly, until the resource returns either null or a zero-length array to signal that there are no more Xids to recover. If the resource has returned all the Xids in the previous `XAResource.recover(TMSTARTRSCAN)` call, then it can either return null or a zero-length array here, or it may also throw `XAER_PROTO`, to indicate that it has finished and forgotten the previous recovery scan. A common `XAResource.recover` implementation problem is ignoring the flags or always returning the same set of Xids on `XAResource.recover(TMNOFLAGS)`. This causes the WebLogic Server transaction manager recovery to loop infinitely, and subsequently fail.
- The WebLogic Server transaction manager `XAResource.recover` with `TMENDRSCAN` flag to end the recovery scan. The resource may return additional Xids.

Resource Health Monitoring

To prevent losing server threads to faulty XAResources, WebLogic Server JTA has an internal resource health monitoring mechanism. A resource is considered active if either there are no pending requests or the result from any of the XAResource pending requests is not `XAER_RMFAIL`. If an XAResource is not active within two minutes, the WebLogic Server transaction manager declares it dead.

Any further requests to the XAResource are shunned, and an `XAER_RMFAIL XAException` is thrown.

The two minute interval can be configured using the `maxXACallMillis` JTAMBean attribute. You can configure `maxXACallMillis` in the `config.xml` file. For example:

```
<Domain>
....
<JTA
   MaxXACallMillis="240000"
/>
....
</Domain>
```

To receive notification from the WebLogic Server transaction manager and to inform the WebLogic Server transaction manager whether it is indeed dead when the resource is about to be declared dead, you can implement `weblogic.transaction.XAResource` (which extends `jakarta.transaction.xa.XAResource`) and register it with the transaction manager. The transaction manager calls the `detectUnavailable` method of the XAResource when it is about to declare it unavailable. If the XAResource returns `true`, then it is not declared unavailable. If the XAResource is indeed unavailable, it can use this opportunity to perform cleanup and re-

Copyright © 2007, 2026, Oracle and/or its affiliates.

registration with the transaction manager. See `weblogic.transaction.XAResource` in the *Java API Reference for Oracle WebLogic Server*.

Jakarta EE Connector Architecture Resource Adapter

Besides registering with the WebLogic Server transaction manager directly, you can also implement the Jakarta EE Connector Architecture resource adapter interfaces. When you deploy the resource adapter, the WebLogic Server Jakarta EE container registers the resource manager's XAResource with the WebLogic Server transaction manager automatically.

See *Developing Resource Adapters for Oracle WebLogic Server*.

Implementation Tips

Learn some tips for exporting and importing transactions with the WebLogic Server transaction manager.

- Sharing the WebLogic Server Transaction Log
- Transaction global properties
- TxHelper.createXid

Sharing the WebLogic Server Transaction Log

The WebLogic Server transaction manager exposes the transaction log to be shared with system applications such as gateways. This provides a way for system applications to take advantage of the box-carring (batching) transaction log optimization of the WebLogic Server transaction manager for fast logging. Note that it is important to release the transaction log records in a timely fashion. (The WebLogic Server transaction manager only removes a transaction log file if all the records in it are released). Failure to do so may result in a large number of transaction log files, and could lead to re-commit of a large number of already committed transactions, or in an extreme case, circular collision and overwriting of transaction log files.

The WebLogic Server transaction manager exposes a transaction logger interface: `weblogic.transaction.TransactionLogger`. It is only available on the server, and it can be obtained with the following steps:

1. Get the server transaction manager:

```
import weblogic.transaction.ServerTransactionManager;
import weblogic.transaction.TxHelper;
ServerTransactionManager stm =
(ServerTransactionManager)TxHelper.getTransactionManager();
```

2. Get the `TransactionLogger`:

```
TransactionLogger tlog = stm.getTransactionLogger();
```

The XAResource's log records must implement the `weblogic.transaction.TransactionLoggable` interface in order to be written to the transaction log. For more information about the `weblogic.transaction.TransactionLogger` interface and usage of the `TransactionLogger` interface, see `weblogic.transaction.TransactionLogger` in the *Java API Reference for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Transaction global properties

A WebLogic Server JTA transaction object is associated with both local and global properties. Global properties are propagated with the transaction propagation context among servers, and are also saved as part of the log record in the transaction log. You can access the transaction global properties as follows:

1. Obtain the transaction object:

```
import weblogic.transaction.Transaction;
import weblogic.transaction.TransactionHelper;
Transaction tx = TransactionHelper.getTransaction();  // Get the transaction
associated with the thread
```

or

```
Transaction tx = TxHelper.getTransaction(xid); // Get the transaction with the given
Xid
```

2. Get or set the properties on the transaction object:

```
tx.setProperty("foo", "fooValue");
tx.getProperty("bar");
```

See `weblogic.transaction.TxHelper` in the *Java API Reference for Oracle WebLogic Server*.

TxHelper.createXid

You can use the `TxHelper.createXid(int formatId, byte[] gtrid, byte[] bqual)` method to `create Xids`, for example, to return to the WebLogic Server transaction manager on recovery.

See `weblogic.transaction.TxHelper` in the *Java API Reference for Oracle WebLogic Server*.

Changes in the Resource Registration Name

This release changes the behavior of the resource registration name for XA data source configurations. In previous releases, the JTA registration name was simply the name of the data source. Now, the registration name is a combination of data source name and domain.

All resources registered with JTA now have a corresponding runtime MBean that exposes XA usage statistics for the resource. This altered (qualified) the JMX `ObjectName` of the MBean, and may impact existing applications that perform a JMX lookup of such a runtime MBean by name. In previous releases, a data source configuration with a name of *`mydatasource`* in domain *`mydomain`* would have a JTA resource runtime MBean registered under the object name:

```
com.bea:ServerRuntime=myserver,Name=mydatasource,Type=TransactionResourceRuntime,
JTARuntime=JTARuntime
```

For this release, the new qualified object name is:

```
com.bea:ServerRuntime=myserver,Name=mydatasource_mydomain,Type=TransactionResourc
eRuntime,JTARuntime=JTARuntime
```

The transaction branch qualifier is also derived from the JTA resource registration name. Any pending transaction branches for XA data sources at the time of upgrade may not be recoverable after upgrade. Oracle recommends that no pending transactions are left pending

Copyright © 2007, 2026, Oracle and/or its affiliates.

in database resources prior to upgrade. Otherwise, any pending database transactions may need to be resolved manually by a database administrator.

This release provides a new system property to disable the qualifying of the registration name:

```
-Dweblogic.jdbc.qualifyRMName=false
```

FAQs

Get to know about the most frequently asked questions related to transactions.

- Why does the XAResource's Xid have a branch qualifier, but not the transaction manager's transaction?

  WebLogic Server JTA transaction objects do not have branch qualifiers (for example, `TxHelper.getTransaction().getXid().getBranchQualifier()` would be null). Since the branch qualifiers are specific to individual resource managers, the WebLogic Server transaction manager only sets the branch qualifiers in the Xids that are passed into XAResource methods.

- What is the `TxHelper.getTransaction()` method used for?

  The WebLogic Server JTA provides the `TxHelper.getTransaction()` API to return the transaction associated with the current thread. However, note that WebLogic Server JTA suspends the transaction context before calling the XAResource methods, so you should only rely on the Xid input parameter to identify the transaction, but not the transaction associated with the current thread.

Additional Documentation about JTA

Additional documentation for a connection-based Resource Usage scenario, which illustrates the JTA interaction between the transaction manager and resource manager is provided in the JTA specification.

Refer to the JTA specification 2.0 Section 4.1 for a connection-based Resource Usage scenario, which illustrates the JTA interaction between the transaction manager and resource manager. The JTA specification is available at https://jakarta.ee/specifications/transactions/2.0/.

Copyright © 2007, 2026, Oracle and/or its affiliates.

14

Participating in Transactions Managed by a Third-Party Transaction Manager

Learn how WebLogic Server participates in distributed transactions coordinated by third-party systems (referred to as foreign transaction managers). The WebLogic Server processing occurs as part of the work of the external transaction. The foreign transaction manager then drives the WebLogic Server transaction manager as part of its commit processing. This is referred to as "importing" transactions into WebLogic Server.

- Overview of Participating in Foreign-Managed Transactions
  The WebLogic Server transaction manager exposes a `javax.transaction.xa.XAResource` implementation using the `weblogic.transaction.InterposedTransactionManager` interface. A foreign transaction manager can access the `InterposedTransactionManager` interface to coordinate the WebLogic Server transaction manager `XAResource` during its commit processing.
- Importing Transactions with the Client Interposed Transaction Manager
  You can use the client interposed transaction manager in WebLogic Server to drive the two-phase commit process for transactions that are coordinated by a third-party transaction manager and include transaction participants within WebLogic Server, such as JMS resources and JDBC resources. The client interposed transaction manager is an implementation of the `javax.transaction.xa.XAResource` interface.
- Importing Transactions with the Server Interposed Transaction Manager
  You can use the server interposed transaction manager in WebLogic Server to drive the two-phase commit process for transactions that are coordinated by a third-party transaction manager and include transaction participants within WebLogic Server, such as JMS resources and JDBC resources. The server interposed transaction manager is an implementation of the `javax.transaction.xa.XAResource` interface.
- Transaction Processing for Imported Transactions
  You can import a foreign transaction into WebLogic Server using the foreign transaction manager or gateway.
- Commit Processing for Imported Transactions
  Learn how foreign transaction manager prepares the interposed transaction manager for the commit process of imported transactions.
- Recovery for Imported Transactions
- Transactions that Span Transaction Manager Systems
  You can configure tight coupling of transaction branches that span different transaction manager systems by selecting the `Enable Tightly Coupled Transactions` attribute on the `Cluster:Configuration:JTA` page in the WebLogic Remote Console.

Overview of Participating in Foreign-Managed Transactions

The WebLogic Server transaction manager exposes a `javax.transaction.xa.XAResource` implementation using the `weblogic.transaction.InterposedTransactionManager` interface. A foreign transaction manager can access the `InterposedTransactionManager` interface to coordinate the WebLogic Server transaction manager `XAResource` during its commit processing.

Copyright © 2007, 2026, Oracle and/or its affiliates.

When importing a transaction from a foreign transaction manager into the WebLogic Server transaction manager, you must register the WebLogic Server interposed transaction manager (ITM) as a subordinate with the foreign transaction manager. The WebLogic Server transaction manager then acts as the coordinator for the imported transaction within WebLogic Server.

WebLogic Server supports two configuration schemes for importing transactions:

- Using a client-side gateway (implemented externally to WebLogic Server) that uses the *client* interposed transaction manager
- Using a server-side gateway implemented on a WebLogic Server instance that uses the *server* interposed transaction manager

Although there are some differences in limitations and in implementation details, the basic behavior is the same for importing transactions in both configurations:

1. Lookup the WebLogic Server transaction manager and register it as an XAResource as necessary in the third-party system.
2. Enlist and delist applicable transaction participants during transaction processing.
3. Send the prepare message to the WebLogic Server transaction manager, which then acts as a subordinate transaction manager and coordinates the prepare phase for transaction participants within WebLogic Server.
4. Send the commit or roll back message to the WebLogic Server transaction manager, which then acts as a subordinate transaction manager and coordinates the second phase of the two-phase commit process for transaction participants within WebLogic Server.
5. Unregister, as necessary.

Importing Transactions with the Client Interposed Transaction Manager

You can use the client interposed transaction manager in WebLogic Server to drive the two-phase commit process for transactions that are coordinated by a third-party transaction manager and include transaction participants within WebLogic Server, such as JMS resources and JDBC resources. The client interposed transaction manager is an implementation of the `javax.transaction.xa.XAResource` interface.

You access the client interposed transaction manager directly from the third-party application, typically from a gateway in the third-party application. The transaction manager in the third-party system then sends the prepare and commit messages to the gateway, which propagates the message to the WebLogic Server transaction manger. The WebLogic Server transaction manager then acts as a subordinate transaction manager and coordinates the transaction participants within WebLogic Server. Figure 14-1 shows the interaction between the two transaction managers and the client-side gateway.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Figure 14-1 Importing Transactions into WebLogic Server Using a Client-Side Gateway

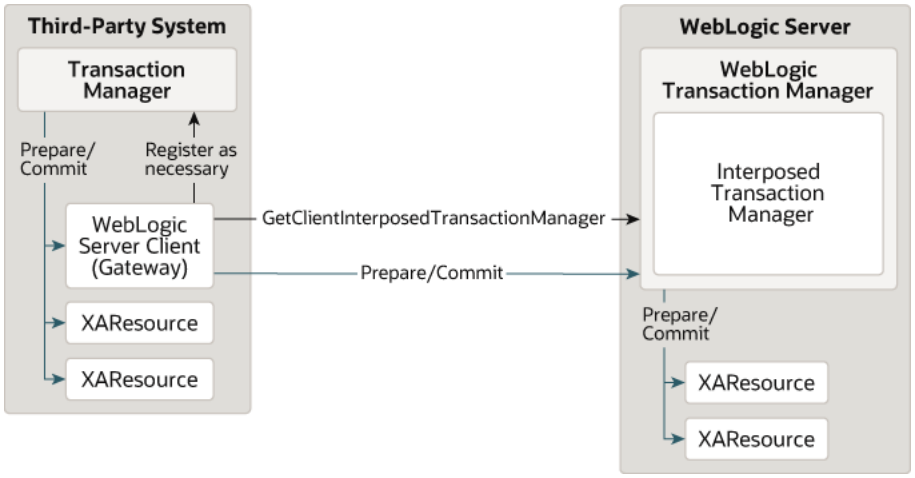
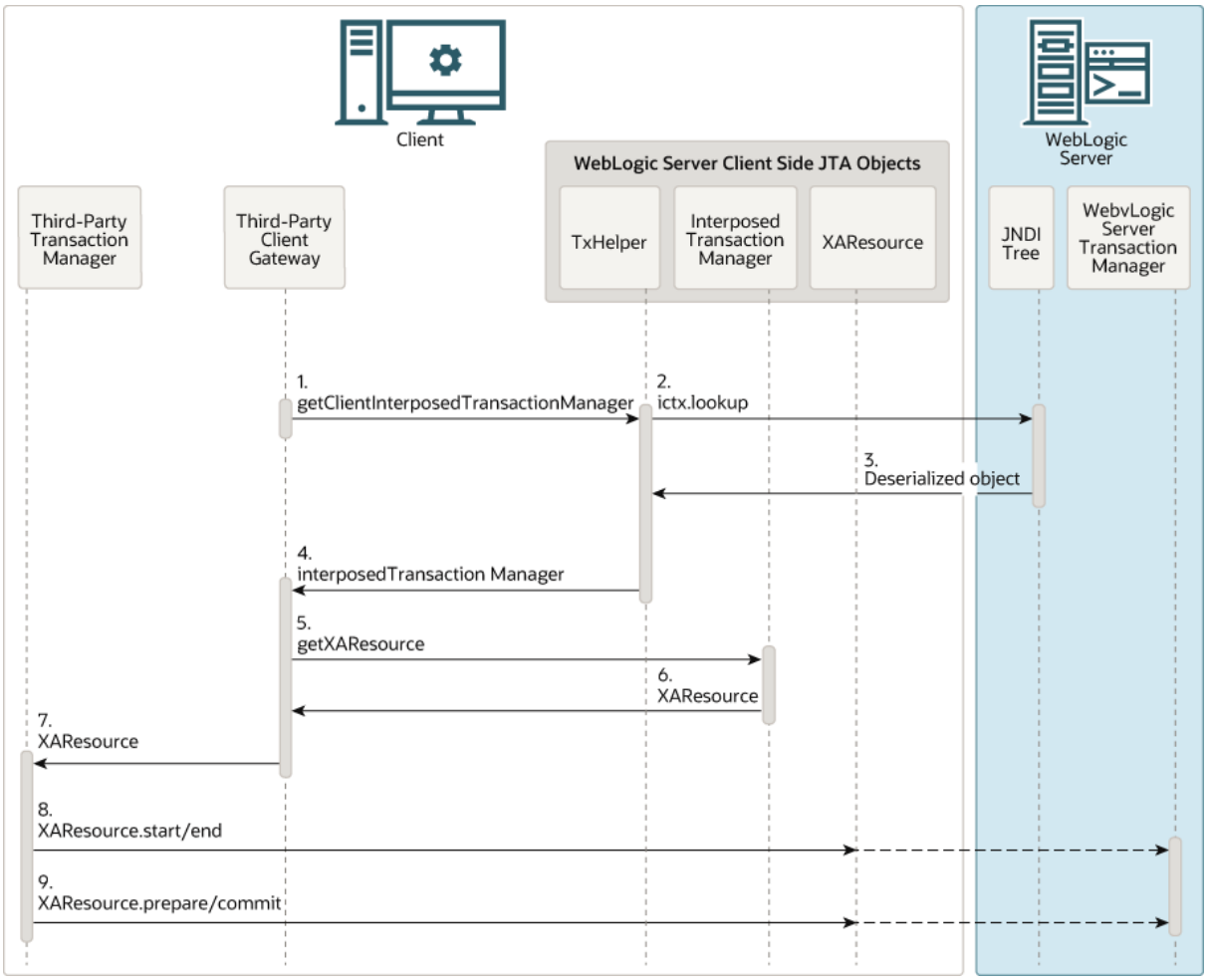


Figure 14-2 shows the flow of interactions between a foreign transaction manager, WebLogic Server client-side JTA objects, and the WebLogic Server transaction manager.

Figure 14-2 State Diagram Illustrating Steps to Import a Transaction Using the Client Interposed Transaction Manager



Copyright © 2007, 2026, Oracle and/or its affiliates.

To access the interposed transaction manager in WebLogic Server using a client-side gateway, you must perform the following steps:

- Get the Client Interposed Transaction Manager
- Get the XAResource from the Interposed Transaction Manager
- Cluster-wide Recovery
- Limitations of the Client Interposed Transaction Manager

Get the Client Interposed Transaction Manager

In a client-side gateway, the you can get the WebLogic server interposed transaction manager's `XAResource` with the `getClientInterposedTransactionManager` method. For example:

```
import javax.naming.Context;
import weblogic.transaction.InterposedTransactionManager;
import weblogic.transaction.TxHelper;
Context initialCtx;
String serverName;
InterposedTransactionManager itm =
TxHelper.getClientInterposedTransactionManager(initialCtx, serverName);
```

The server name parameter is the name of the server that acts as the interposed transaction manager for the foreign transaction. When the foreign transaction manager performs crash recovery, it must contact the same WebLogic Server server to obtain the list of in-doubt transactions that were previously imported into WebLogic Server.

See `weblogic.transaction.TxHelper` in the *Java API Reference for Oracle WebLogic Server*.

Get the XAResource from the Interposed Transaction Manager

After you get the interposed transaction manager, you must get the XAResource object associated with the interposed transaction manager:

```
import javax.transaction.xa.XAResource;
XAResource xar = itm.getXAResource();
```

Cluster-wide Recovery

You can configure cluster-wide transaction recovery of distributed transactions across all the interposed transaction managers of a cluster by selecting the `Enable Cluster-Wide Recovery` attribute on the Configure Domain JTA Options page in the *Oracle WebLogic Remote Console Online Help*.

When enabled, each interposed transaction manager in a WebLogic cluster is aware of the transaction distribution across the entire cluster. This allows the interposed transaction manager on each cluster member to determine if it should handle a given XA call or forward it to the appropriate interposed transaction manager on another cluster member. In addition:

- An `XAResource.recover` call on a single ITM within a WebLogic cluster returns a list of in-doubt transactions of all interposed transaction manager 's within the cluster.
- If a cluster member fails, the interposed transaction manager is migrated to another cluster member within the same cluster.
- If a WebLogic distributed destination is present on a cluster, messages could be load balanced to other cluster members other than the member hosting the interposed

Copyright © 2007, 2026, Oracle and/or its affiliates.

transaction manager. If this occurs, the interposed transaction manager transparently handles any XA calls and forwards them to the correct cluster member instance. In the situation that the cluster member that hosts the destination fails. The interposed transaction manager transparently handles the potential failover of that destination member.

Limitations of the Client Interposed Transaction Manager

Note the following limitations when importing transactions using a client-side gateway:

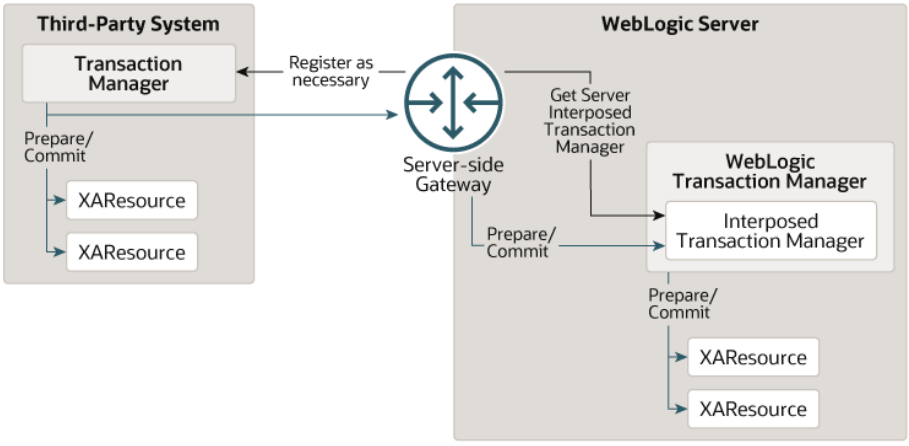
- You cannot use the `TxHelper.getServerInterposedTransactionManager()` method in client-side gateways.

Importing Transactions with the Server Interposed Transaction Manager

You can use the server interposed transaction manager in WebLogic Server to drive the two-phase commit process for transactions that are coordinated by a third-party transaction manager and include transaction participants within WebLogic Server, such as JMS resources and JDBC resources. The server interposed transaction manager is an implementation of the `javax.transaction.xa.XAResource` interface.

You access the server interposed transaction manager by creating a server-side gateway on WebLogic Server and then accessing the gateway from a third-party system. The transaction manager in the third-party system then sends the prepare and commit messages to the server-side gateway, which propagates the message to the WebLogic Server transaction manager. The WebLogic Server transaction manager then acts as a subordinate transaction manager and coordinates the transaction participants within WebLogic Server. Figure 14-3 shows the interaction between the two transaction managers and the server-side gateway.

Figure 14-3 Importing Transactions into WebLogic Server Using a Server-Side Gateway



To access the interposed transaction manager in WebLogic Server using a server-side gateway, you must perform the following steps:

- Get the Server Interposed Transaction Manager

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Limitations of the Server Interposed Transaction Manager

Get the Server Interposed Transaction Manager

In a server-side gateway, you can get the interposed transaction manager's XAResource as follows:

```
import javax.naming.Context;
import weblogic.transaction.InterposedTransactionManager;
import weblogic.transaction.TxHelper;
InterposedTransactionManager itm = TxHelper.getServerInterposedTransactionManager();
```

See `weblogic.transaction.TxHelper` in the *Java API Reference for Oracle WebLogic Server*.

After you get the interposed transaction manager, you must get the `XAResource`. See Get the XAResource from the Interposed Transaction Manager.

Limitations of the Server Interposed Transaction Manager

Note the following limitations when importing transactions using a server-side gateway:

- Do not use the `TxHelper.getClientInterposedTransactionManager()` method in a server-side gateway on a WebLogic Server server. Doing so causes performance issues.
- You can only use *one* WebLogic Server server interposed transaction manager at a time. Do not use multiple server interposed transaction managers (on the same thread) to import transactions at the same time. (See Transaction Processing for Imported Transactions for more information about this limitation and how transactions are processed with the WebLogic Server interposed transaction manager.)

Transaction Processing for Imported Transactions

You can import a foreign transaction into WebLogic Server using the foreign transaction manager or gateway.

To import a foreign transaction into WebLogic Server, the foreign transaction manager or gateway can do the following:

```
xar.start(foreignXid, TMNOFLAGS);
```

This operation associates the current thread with the imported transaction. All subsequent calls made to other servers propagate the imported WebLogic Server transaction, until the transaction is disassociated from the thread.

> **Note**
>
> The flag is ignored by the WebLogic Server transaction manager. If the foreign Xid has been imported previously on the same WebLogic Server server, WebLogic Server associates the current thread with the previously imported WebLogic Server transaction.

To disassociate the imported transaction from the current thread, the foreign transaction manager or gateway should do the following:

```
xar.end(foreignXid, TMSUCCESS);
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Note that the WebLogic Server transaction manager ignores the flag.

- Transaction Processing Limitations for Imported Transactions

Transaction Processing Limitations for Imported Transactions

Note the following processing limitations and behavior for imported transactions:

- After a WebLogic Server transaction is started, the gateway cannot call start again on the same thread. With a client-side gateway, you can only call `xar.start` on one client interposed transaction manager at a time. Attempting to call `xar.start` on another client interposed transaction manager (before `xar.end` was called on the first one) throws an `XAException` with `XAER_RMERR`. With a server-side gateway, attempting to call `xar.start` on a client or server interposed transaction manager also throws a `XAException` with `XAER_RMERR` if there is an active transaction associated with the current thread.
- The WebLogic Server interposed transaction manager's `XAResource` exhibits loosely-coupled transaction branching behavior on different WebLogic Server servers. That is, if the same foreign Xid is imported on different WebLogic Server servers, they are imported to different WebLogic Server transactions.
- The WebLogic Server transaction manager does not flatten the transaction tree, for example, the imported transaction of a previously exported WebLogic Server transaction are in a separate branch from the original WebLogic Server transaction.
- A foreign transaction manager should ensure that all foreign Xids that are imported into WebLogic Server are unique and are not reused within the sum of the transaction abandon timeout period and the transaction timeout period. Failure to do so may result in log records that are never released in the WebLogic Server transaction manager. This could lead to inefficient crash recovery.

Commit Processing for Imported Transactions

Learn how foreign transaction manager prepares the interposed transaction manager for the commit process of imported transactions.

The foreign transaction manager should drive the interposed transaction manager in the 2PC protocol as it does the other `XAResources`. Note that the `beforeCompletion` callbacks registered with the WebLogic Server JTA (for example, the EJB container) are called when the foreign transaction manager prepares the interposed transaction manager's `XAResource`. The `afterCompletion` callbacks are called during `XAResource.commit` or `XAResource.rollback`.

The WebLogic Server interposed transaction manager honors the `XAResource` contract as described in the Java Transaction API at `https://jakarta.ee/specifications/transactions/2.0/`.

- Once prepared by a foreign transaction manager, the WebLogic Server interposed transaction manager waits persistently for a commit or rollback outcome from the foreign transaction manager until the transaction abandon timeout expires.
- The WebLogic Server interposed transaction manager remembers heuristic outcomes persistently until being told to forget about the transaction by the foreign transaction manager or until transaction abandon timeout.

The WebLogic Server transaction manager logs a `prepare` record for the imported transaction after all the WebLogic Server participants are successfully prepared. If there are multiple WebLogic Server participants for the imported transaction, the transaction manager logs a prepare record even if the `XAResource.commit` is a one-phase commit.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Recovery for Imported Transactions

During the crash recovery of the foreign transaction manager, the foreign transaction manager must get the `XAResource` of the WebLogic Server interposed transaction manager again, and call recover on it. The WebLogic Server interposed transaction manager then returns the list of prepared or heuristically completed transactions. The foreign transaction manager should then resolve those in-doubt transactions: either commit or rollback the prepared transactions, and call forget on the heuristically completed transactions.

Transactions that Span Transaction Manager Systems

You can configure tight coupling of transaction branches that span different transaction manager systems by selecting the `Enable Tightly Coupled Transactions` attribute on the `Cluster:Configuration:JTA` page in the WebLogic Remote Console.

When he `Enable Tightly Coupled Transactions` attribute is enabled, WebLogic Server uses the transaction identifier of a transaction imported by the interposed transaction manager for XA calls rather than an internally mapping a transaction id. This tighter coupling of transaction managers typically improves performance and applies to inter-domain WebLogic transactions and transactions imported from Tuxedo .

Note

- Oracle does not support tightly coupled transactions with other external transaction processing systems for which interoperability is enabled using WSAT. However, for EJBs and applications within a single cluster that uses the same transaction id, tight coupling can be enabled. For more information about WSAT, see Using Web Services Atomic Transactions in Developing JAX-WS Web Services for Oracle WebLogic Server.

If a transaction between WebLogic and Tuxedo resources uses a GridLink Data Source with GridLink Affinity enabled, the XA Affinity context is automatically used for the transaction.

Copyright © 2007, 2026, Oracle and/or its affiliates.

15

Troubleshooting Transactions

Learn about the troubleshooting tools and tasks that are used in determining why transactions fail and deciding what actions to take to correct the problem.

- Overview
  WebLogic Server includes the ability to monitor currently running transactions and ensure that adequate information is captured in the case of heuristic completion. It also provides the ability to monitor performance of database queries, transactional requests, and bean methods.
- Troubleshooting Tools
  WebLogic Server provides aids like exceptions, transaction identifier, transaction status, and so on for transaction troubleshooting.

Overview

WebLogic Server includes the ability to monitor currently running transactions and ensure that adequate information is captured in the case of heuristic completion. It also provides the ability to monitor performance of database queries, transactional requests, and bean methods.

Troubleshooting Tools

WebLogic Server provides aids like exceptions, transaction identifier, transaction status, and so on for transaction troubleshooting.

This topic has following sections:

- Exceptions
- Transaction Identifier
- Transaction Name and Properties
- Transaction Status
- Transaction Statistics
- Transaction Monitoring
- Debugging JTA Resources

Exceptions

WebLogic JTA supports all standard JTA exceptions. For more information about standard JTA exceptions, see the API Javadoc for the following packages:

- jakarta.transaction
- javax.transaction.xa

In addition to the standard JTA exceptions, WebLogic Server provides the class `weblogic.transaction.RollbackException`. This class extends `jakarta.transaction.RollbackException` and preserves the original reason for a rollback.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Before rolling a transaction back, or before setting it to `rollbackonly`, an application can supply a reason for the rollback. All rollbacks triggered inside the transaction service set the reason (for example, timeouts, XA errors, unchecked exceptions in `beforeCompletion`, or inability to contact the transaction manager). Once set, the reason cannot be overwritten.

When a transaction is logged in Database using TLOG store and SQL query, if it takes more than 30 seconds then the `SystemException: Transaction could not be logged`, is thrown. To solve this issue, look at database performance and Automatic Workload Repository reports to understand the cause of slow down and accordingly perform database tuning.

Transaction Identifier

The Transaction Service assigns a transaction identifier (`Xid`) to each transaction. This ID can isolate information about a specific transaction in a log file. You can retrieve the transaction identifier using the `getXID` method in the `weblogic.transaction.Transaction` interface. For detailed information on methods for getting the transaction identifier, see `weblogic.transaction.Transaction` in the *Java API Reference for Oracle WebLogic Server*.

Transaction Name and Properties

WebLogic JTA provides extensions to `jakarta.transaction.Transaction` that support transaction naming and user-defined properties. These extensions are included in the `weblogic.transaction.Transaction` interface.

The transaction name indicates a type of transaction (for example, funds transfer or ticket purchase) and should not be confused with the transaction ID, which identifies a unique transaction on a server. The transaction name makes it easier to identify a transaction type in the context of an exception or a log file.

User-defined properties are key/value pairs, where the key is a string identifying the property and the value is the current value assigned to the property. Transaction property values must be objects that implement the `Serializable` interface. You manage properties in your application using the set and get methods defined in the `weblogic.transaction.Transaction` interface. Once set, properties stay with a transaction during its entire lifetime and are passed between machines as the transaction travels through the system. Properties are saved in the transaction log, and are restored during crash recovery processing. If a transaction property is set more than once, the latest value is retained.

For detailed information on methods for setting and getting the transaction name and transaction properties, see `weblogic.transaction.Transaction` in the *Java API Reference for Oracle WebLogic Server*.

Transaction Status

The Java Transaction API provides transaction status codes using the `jakarta.transaction.Status` class. Use the `getStatusAsString` method in `weblogic.transaction.Transaction` to return the status of the transaction as a string. The string contains the major state as specified in `jakarta.transaction.Status` with an additional minor state (such as `logging` or `pre-preparing`).

Transaction Statistics

Transaction statistics are provided for all transactions handled by the transaction manager on a server. These statistics include the number of total transactions, transactions with a specific outcome (such as committed, rolled back, or heuristic completion), rolled back transactions by

Copyright © 2007, 2026, Oracle and/or its affiliates.

reason, and the total time that transactions were active. For detailed information about transaction statistics, see View Transaction Statistics in the *Oracle WebLogic Remote Console Online Help*.

Transaction Monitoring

The Remote Console enables you to monitor transactions. Monitoring tasks are performed at the server level. Transaction statistics are displayed for a specific server.

Debugging JTA Resources

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

- Enabling Debugging
- Enable Debugging Using the Command Line
- Enable Debugging Using the WebLogic Remote Console
- Enable Debugging Using the WebLogic Scripting Tool
- Changes to the config.xml File
- JTA Debugging Scopes

Enabling Debugging

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

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

Enable Debugging Using the Command Line

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

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

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

Enable Debugging Using the WebLogic Remote Console

Use the WebLogic Remote Console to set the debugging values:

1. Log into **WebLogic Remote Console** and navigate to **Edit Tree**.
2. In the left pane, expand **Environment** and select **Servers**.
3. Click the server which you want to enable or disable debugging to view the settings page for that server.
4. Click the **Debug** tab.
5. Select the check box for the debug attributes you want to modify.
6. Click Save.
7. Click the **Shopping Cart** icon and select **Commit Changes**.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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('DebugJDBCJTA','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('DebugJDBCJTA','true')");
              interpreter.exec("save()");
              interpreter.exec("activate()");

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

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Changes to the config.xml File

Changes in debugging characteristics, through the WebLogic Remote Console, or WLST, or command line are persisted in the `config.xml` file. See Example 15-1:

Example 15-1 Example Debugging Stanza for JTA

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

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

JTA Debugging Scopes

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

You can enable the following registered debugging scopes for JTA:

- `DebugJDBCJTA` (scope weblogic.jdbc.transaction) - not currently used.
- `DebugJTAXA` (scope weblogic.transaction.xa) - traces for XA resources.
- `DebugJTANonXA` (scope weblogic.transaction.nonxa) - traces for non-XA resources.
- `DebugJTAXAStackTrace` (scope weblogic.transaction.stacktrace) - detailed tracing that prints stack traces at various critical locations.
- `DebugJTARMI` (scope weblogic.transaction.rmi) - not currently used.
- `DebugJTA2PC` (scope weblogic.transaction.twopc) - traces all two-phase commit operations.
- `DebugJTA2PCStackTrace` (scope weblogic.transaction.twopcstacktrace) - detailed two-phase commit tracing that prints stack traces.
- `DebugJTATLOG` (scope weblogic.transaction.tlog) - traces transaction logging information.
- `DebugJTAJDBC` (scope weblogic.transaction.jdbc, weblogic.jdbc.transaction) - traces information about reading/writing JTA records.
- `DebugJTARecovery` (scope weblogic.transaction.recovery) - traces recovery information.
- `DebugJTAGateway` (scope weblogic.transaction.gateway) - traces information about imported transactions.
- `DebugJTAGatewayStackTrace` (scope weblogic.transaction.gatewaystacktrace) - stack traces related to imported transactions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `DebugJTANaming` (scope weblogic.transaction.naming) - traces transaction naming information.
- `DebugJTANamingStackTrace` (scope weblogic.transaction.namingstacktrace) - traces transaction naming information.
- `DebugJTAResourceHealth` (scope weblogic.transaction.resourcehealth) - traces information about XA transaction resource health.
- `DebugJTAMigration` (scope weblogic.transaction.migration) - traces information about Transaction Log migration.
- `DebugJTALifecycle` (scope weblogic.transaction.lifecycle) - traces information about the transaction server lifecycle (initialization, suspension, resuming, and shutdown).
- `DebugJTALLR` (scope weblogic.transaction.llr) - traces all Logging Last Resource operations.
- `DebugJTAHealth` (scope weblogic.transaction.health) - traces information about transaction subsystem health.
- `DebugJTATransactionName` (scope weblogic.transaction.name) - traces transaction names.
- `DebugJTAResourceName` (scope weblogic.transaction.resourcename) - traces transaction resource names.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Index

Copyright © 2007, 2026, Oracle and/or its affiliates.